Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements for some of materials: Indy Gupta and Nikita Borisov

Logistics

- MP0 is due today at 11:59pm.

- Reminder to share your name when you speak up in class.

- Feb 7 lecture was not recorded! Please see my Campuswire post for links to last year's videos covering the same topic.

Today's agenda

- Global State (contd.)
  - Chapter 14.5
- Multicast
  - Chapter 15.4

Today's agenda

- **Global State (contd.)**
  - Chapter 14.5
- **Multicast**
  - Chapter 15.4

Global Snapshot Summary

- The ability to calculate global snapshots in a distributed system is very important.
- But don't want to interrupt running distributed application.
- Chandy-Lamport algorithm calculates global snapshot.
- Obeys causality (creates a consistent cut).

Chandy-Lamport Algorithm: Properties

- Any run of the Chandy-Lamport Global Snapshot algorithm creates a consistent cut.
- Let $\mathbf{e}_i$ and $\mathbf{e}_j$ be events occurring at $\mathbf{p}_i$ and $\mathbf{p}_j$, respectively such that
  - $\mathbf{e}_i \rightarrow \mathbf{e}_j$ ($\mathbf{e}_i$ happens before $\mathbf{e}_j$)
- The snapshot algorithm ensures that

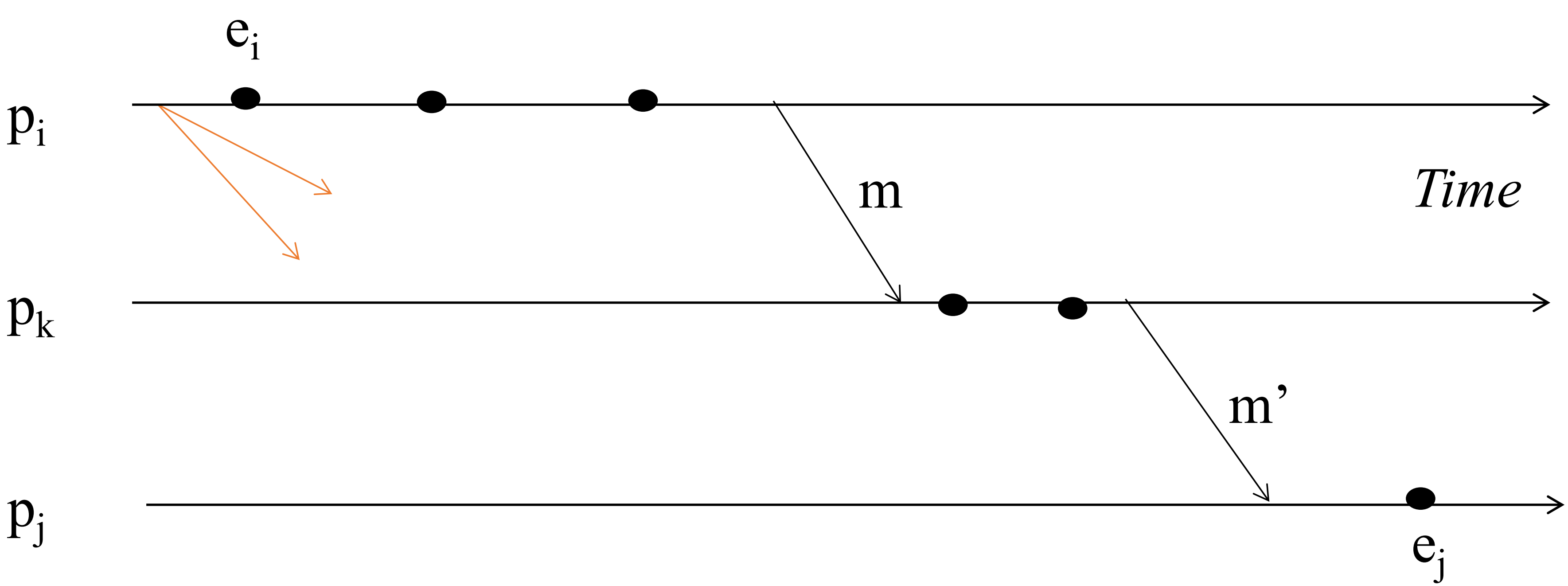
  if $\mathbf{e}_j$ is in the cut then $\mathbf{e}_i$ is also in the cut.

- That is: if $\mathbf{e}_j \rightarrow$ < $\mathbf{p}_j$ records its state>, then

  it must be true that $\mathbf{e}_i \rightarrow$ <$\mathbf{p}_i$ records its state>.

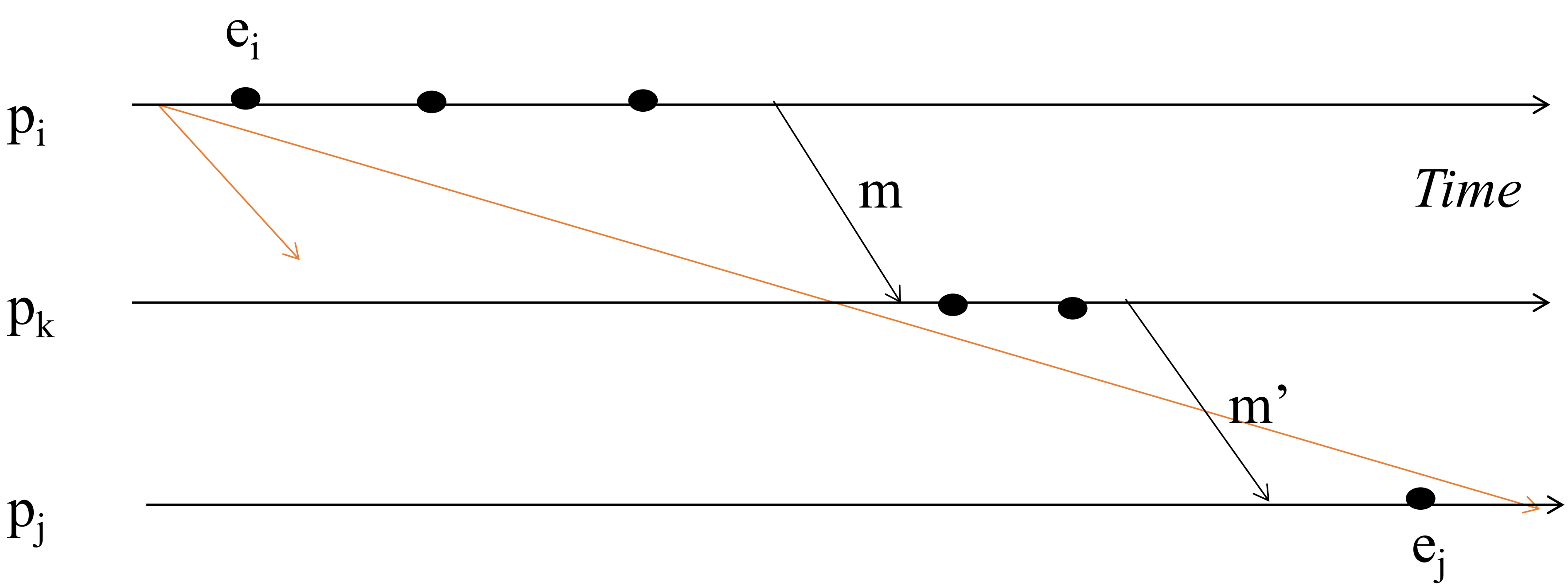
Chandy-Lamport Algorithm: Properties

- If $\mathbf{e_j} \rightarrow$ < $\mathbf{p_j}$ records its state>, then
  it must be true that $\mathbf{e_i} \rightarrow$ <$\mathbf{p_i}$ records its state>.
- By contradiction, *suppose* $\mathbf{e_j} \rightarrow$ *< p_j records its state>, and <p_i records its state>* $\rightarrow \mathbf{e_i}$.



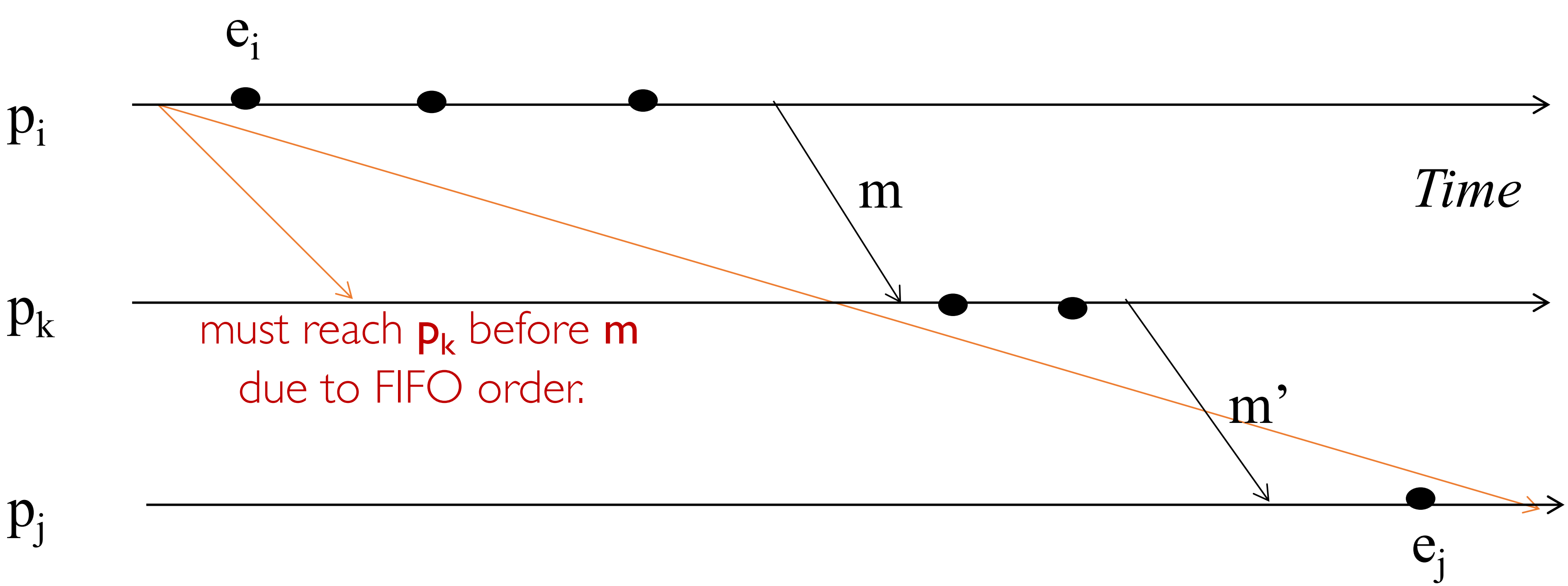
Chandy-Lamport Algorithm: Properties

- If $\mathbf{e_j}$ → < $\mathbf{p_j}$ records its state>, then it must be true that $\mathbf{e_i}$ → <$\mathbf{p_i}$ records its state>.
- By contradiction, *suppose* $\mathbf{e_j}$ *→ <* p_j *records its state>, and <*p_i *records its state>* → $\mathbf{e_i}$*.*



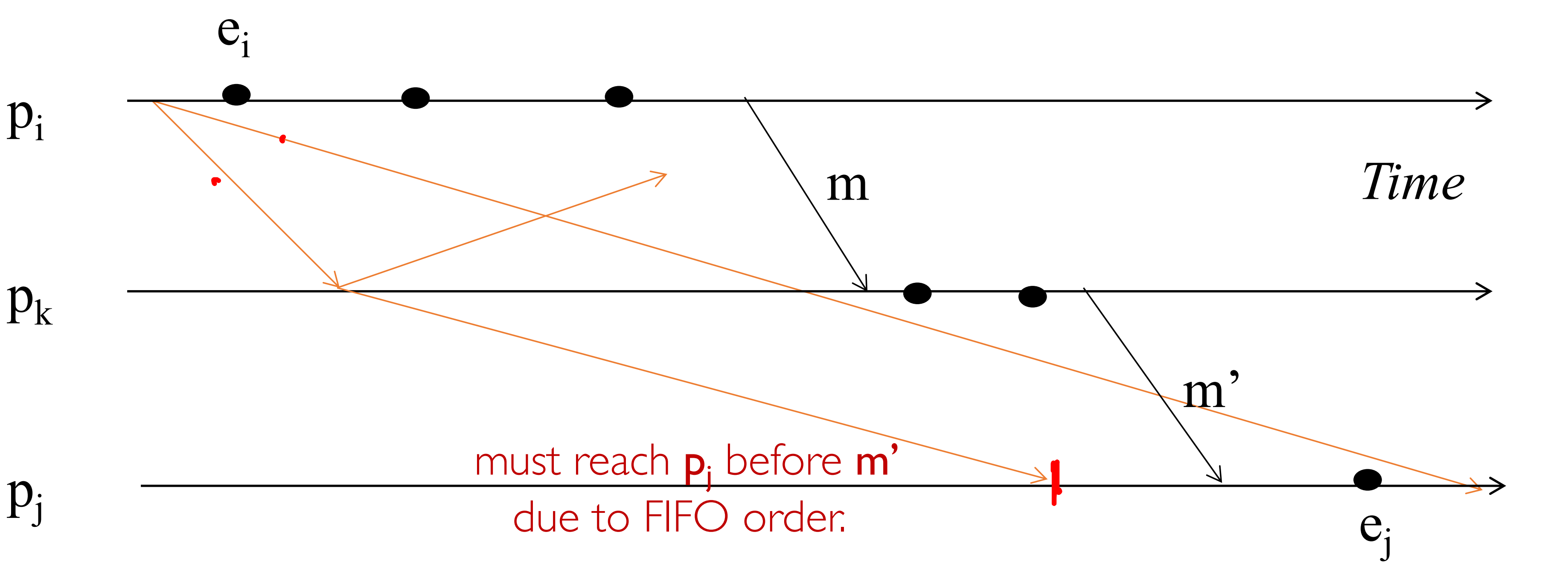
Chandy-Lamport Algorithm: Properties

- If $\mathbf{e_j} \rightarrow$ < $\mathbf{p_j}$ records its state>, then
  it must be true that $\mathbf{e_i} \rightarrow$ <$\mathbf{p_i}$ records its state>.
- By contradiction, *suppose* $\mathbf{e_j} \rightarrow$ *< $\mathbf{p_j}$ records its state>, and <$\mathbf{p_i}$ records its state>* $\rightarrow$ $\mathbf{e_i}$.



Chandy-Lamport Algorithm: Properties

- If $\mathbf{e_j}$ → < $\mathbf{p_j}$ records its state>, then it must be true that $\mathbf{e_i}$ → <$\mathbf{p_i}$ records its state>.
- By contradiction, *suppose* $\mathbf{e_j}$ *→ <* $\mathbf{p_j}$ *records its state>, and <*$\mathbf{p_i}$ *records its state> →* $\mathbf{e_i}$*.*



Chandy-Lamport Algorithm: Properties

- If $\mathbf{e_j}$ → < $\mathbf{p_j}$ records its state>, then it must be true that $\mathbf{e_i}$ → <$\mathbf{p_i}$ records its state>.
- By contradiction, *suppose* $\mathbf{e_j}$ → *<* $\boldsymbol{p_j}$ *records its state>, and <*$\boldsymbol{p_i}$ *records its state>* → $\mathbf{e_i}$.
- Consider the path of app messages (through other processes) that go from $\mathbf{e_i}$ to $\mathbf{e_j}$.
- Due to FIFO ordering, markers on each link in above path will precede regular app messages.
- Thus, since <$\mathbf{p_i}$ records its state> → $\mathbf{e_i}$, it must be true that $\mathbf{p_j}$ received a marker before $\mathbf{e_j}$.
- Thus $\mathbf{e_j}$ is not in the cut => contradiction.

Why capture global snapshots?

- *Checkpointing* the system state.
- Reasoning about unreferenced objects (for garbage collection).
- Distributed debugging.
- Can be used to detect **global properties.**
    - **Safety vs. Liveness**

Revisions: notations and definitions

- For a process p_i , where events $e_i^1, \ldots$ occur:

  history(p_i) = $h_i = <e_i^1, \ldots >$

  prefix history(p_i^k) = $h_i^k = <e_i^1, \ldots, e_i^k >$

  s_i^k : p_i's state immediately after k^{th} event.

- For a set of processes $<p_1, p_2, p_3, \ldots, p_n>$:

  global history: $H = \cup_i (h_i)$

  a cut $C \subseteq H = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_3}$

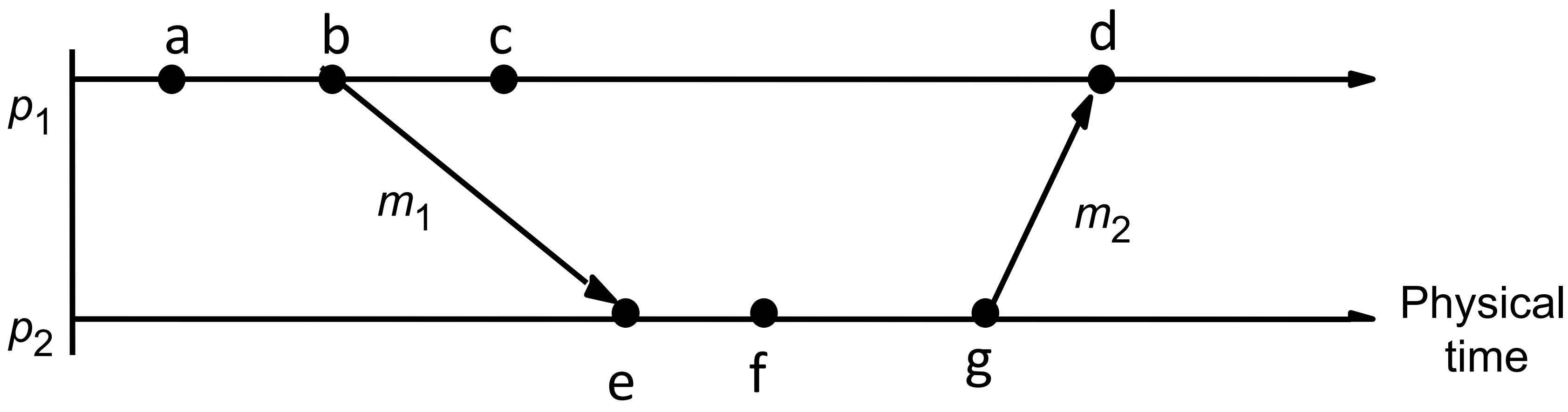
  the frontier of $C = \{e_i^{c_i}, i = 1,2, \ldots n\}$

  global state S that corresponds to cut $C = \cup_i (s_i^{c_i})$

More notations and definitions

- A **run** is a total ordering of events in H that is consistent with each $\mathbf{h_i}$'s ordering.
- A **linearization** is a run consistent with happens-before ($\rightarrow$) relation in H.

Example



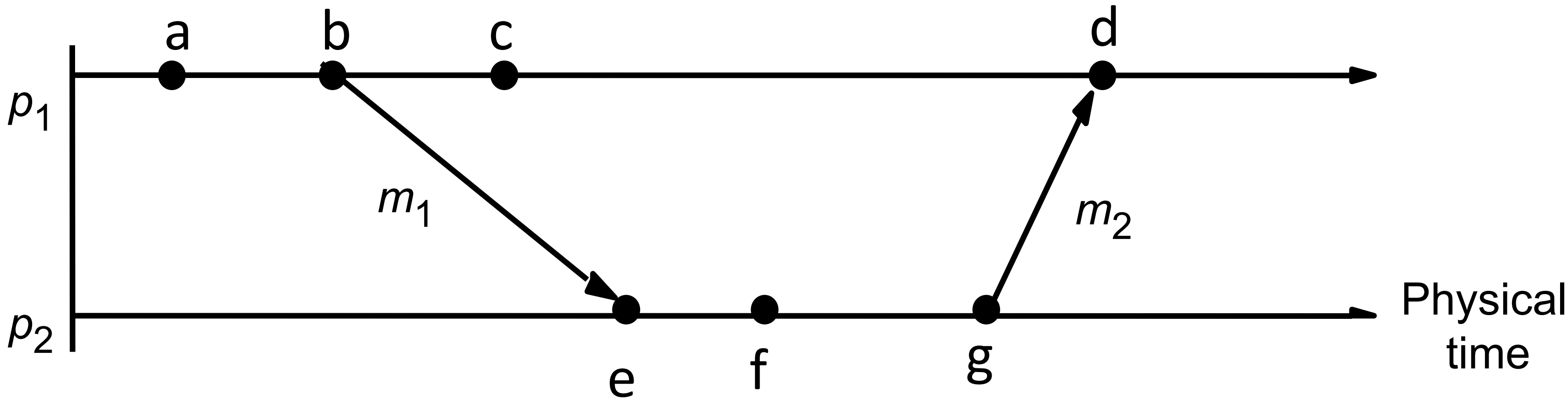
Order at p_1: < a, b, c, d >   Order at p_2: < e, f,g>

Causal order across p_1 and p_2: < b, e> , <g , d >

Run: < a, b, c, d , e, f, g>

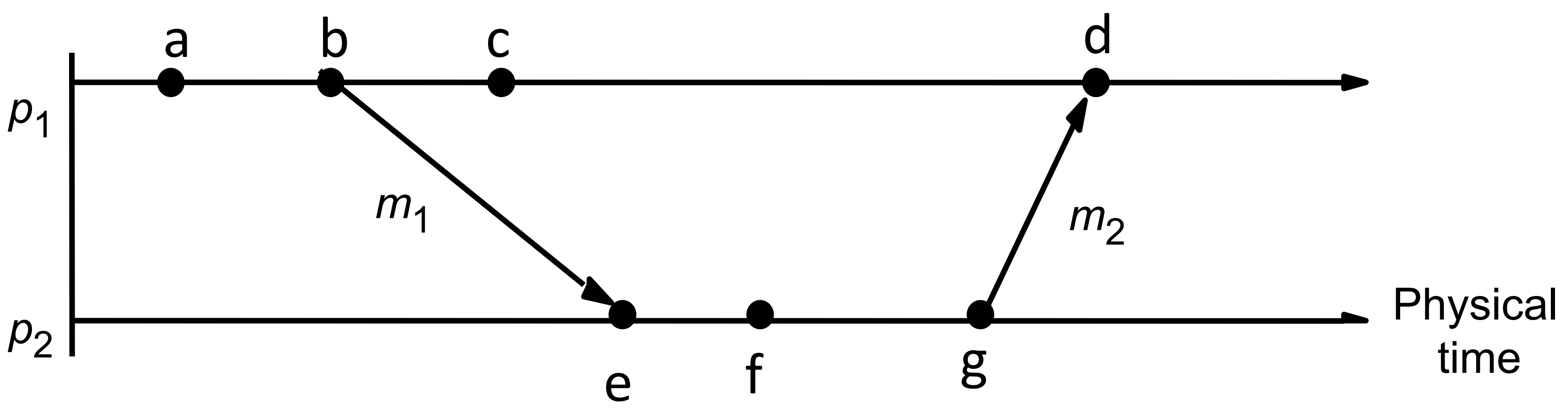
Linearization: <a, b, c, e, f, g , d >

Example



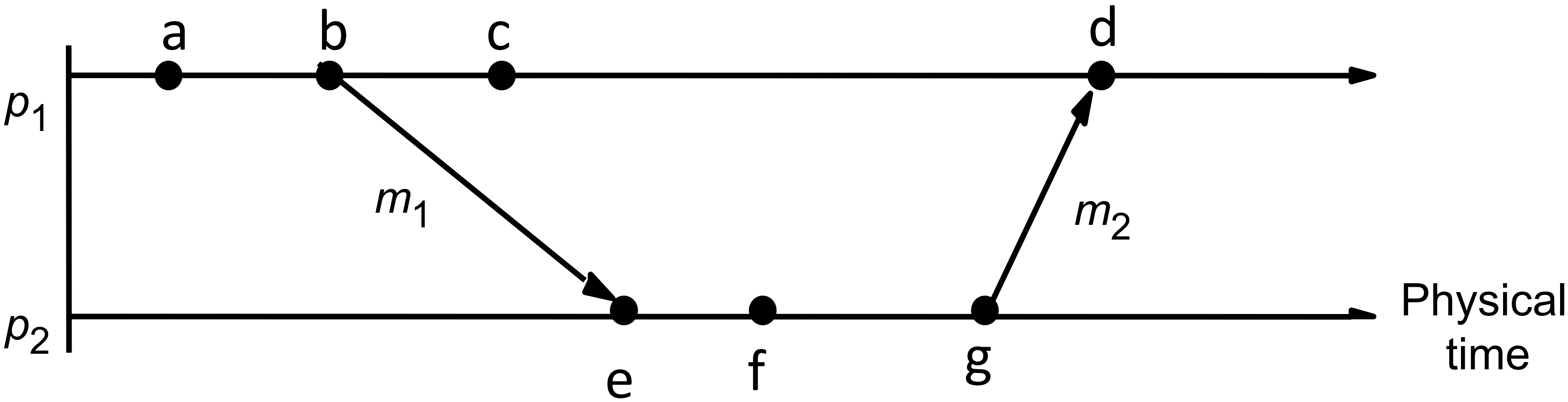
< a, b, e, f, c, g , d >:

Example



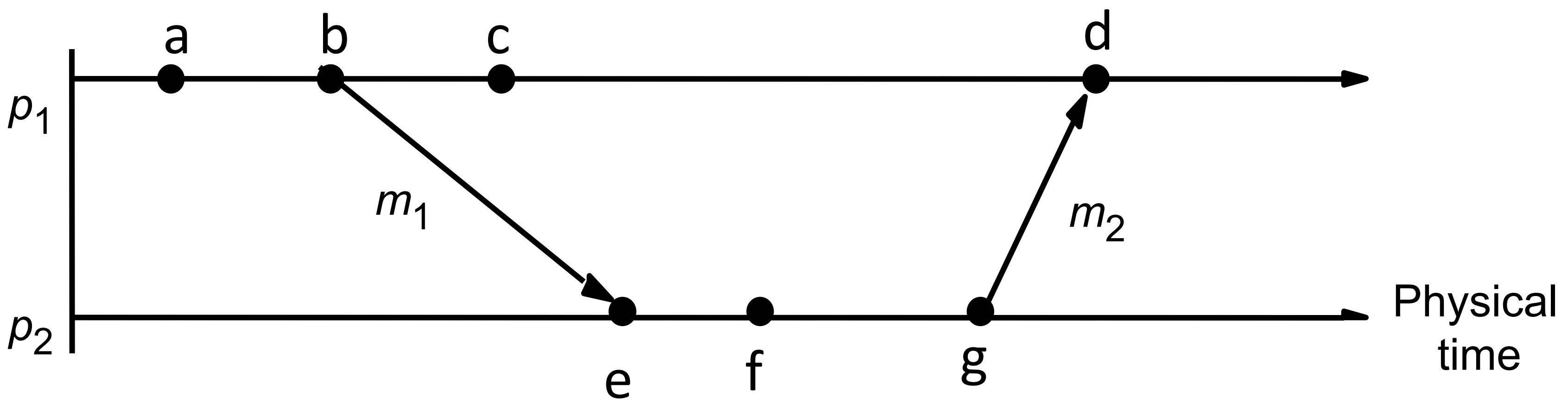
< a, b, e, f, c, g , d >: Linearization

Example



< a, f, e , b, c, g , d >:

Example

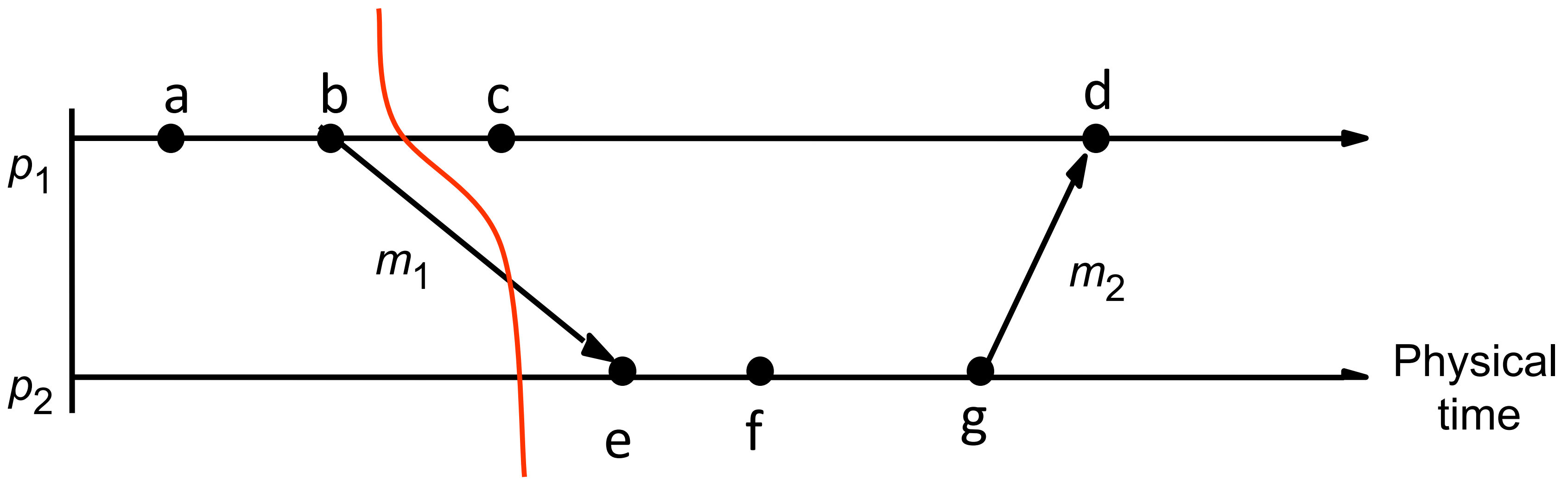


< a, f, e , b, c, g , d >: Not even a run

More notations and definitions

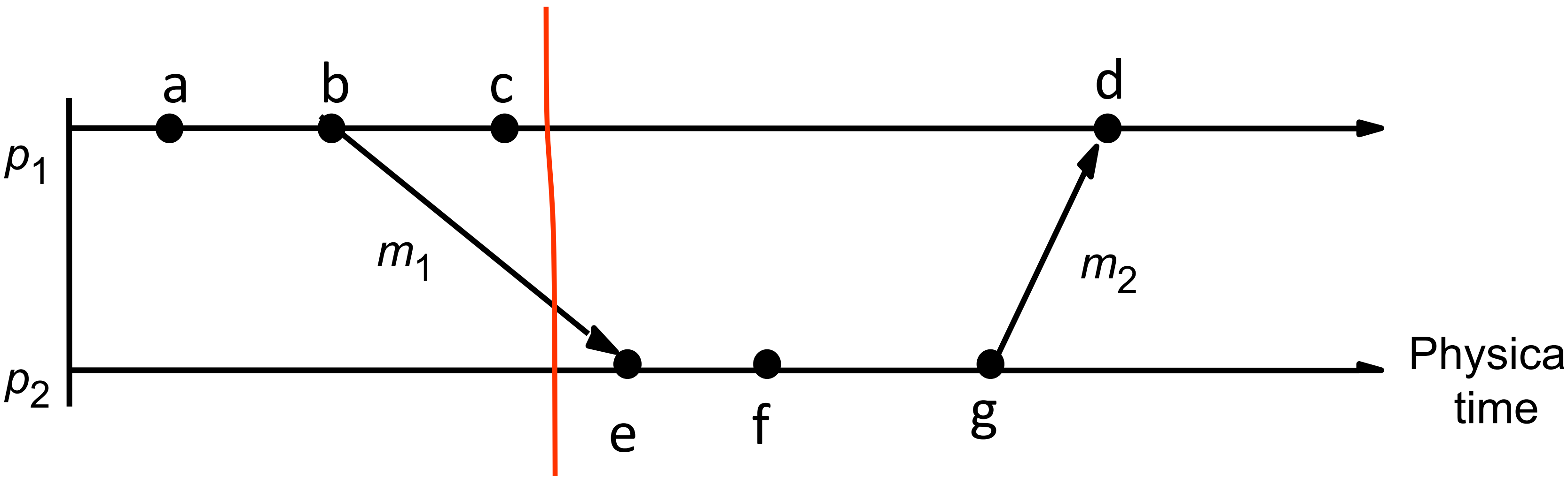
- A **run** is a total ordering of events in H that is consistent with each $\mathbf{h_i}$'s ordering.
- A **linearization** is a run consistent with happens-before ($\rightarrow$) relation in H.
- Linearizations pass through consistent global states.

Example



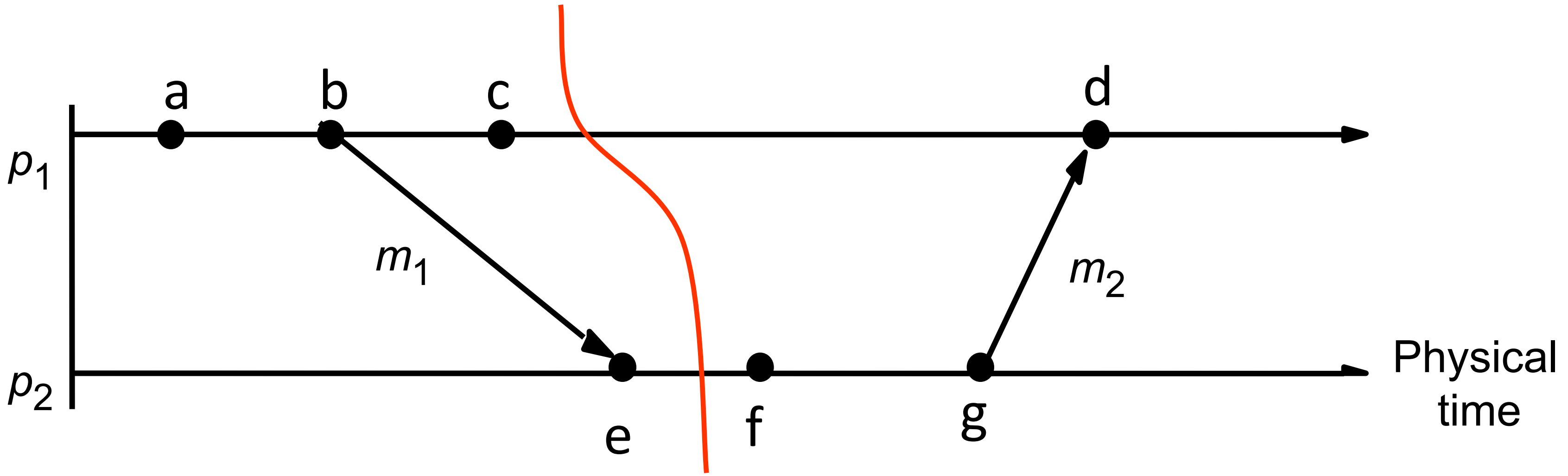
Linearization: < a, b, c, e, f, g , d >

Example



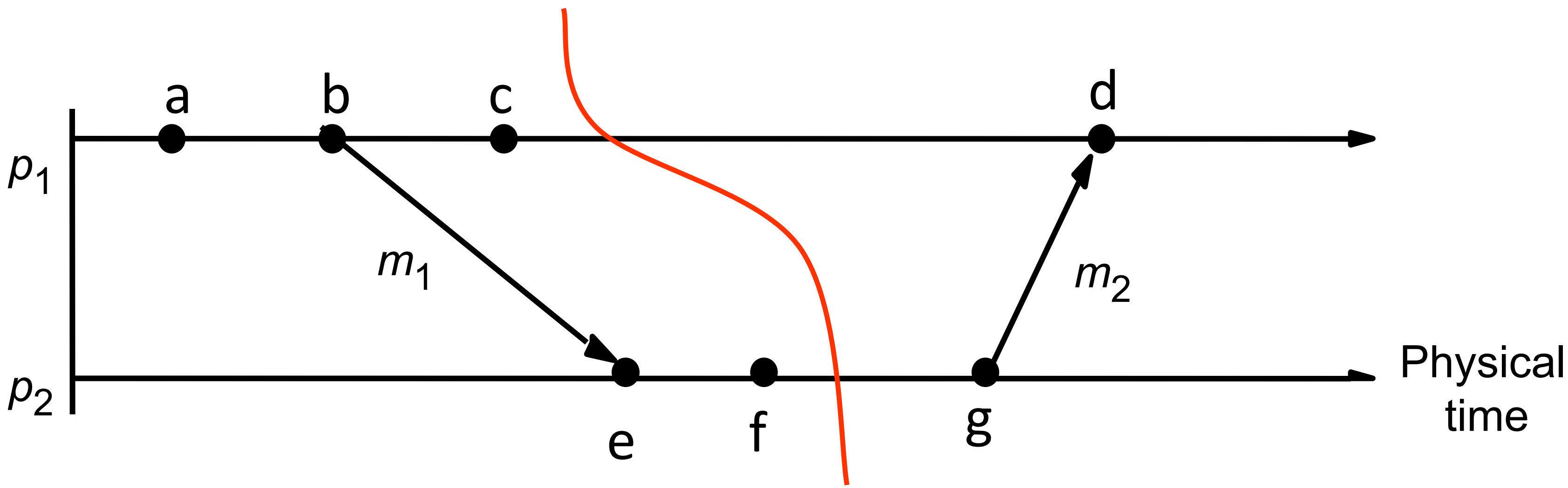
Linearization: < a, b, c, e, f, g , d >

Example



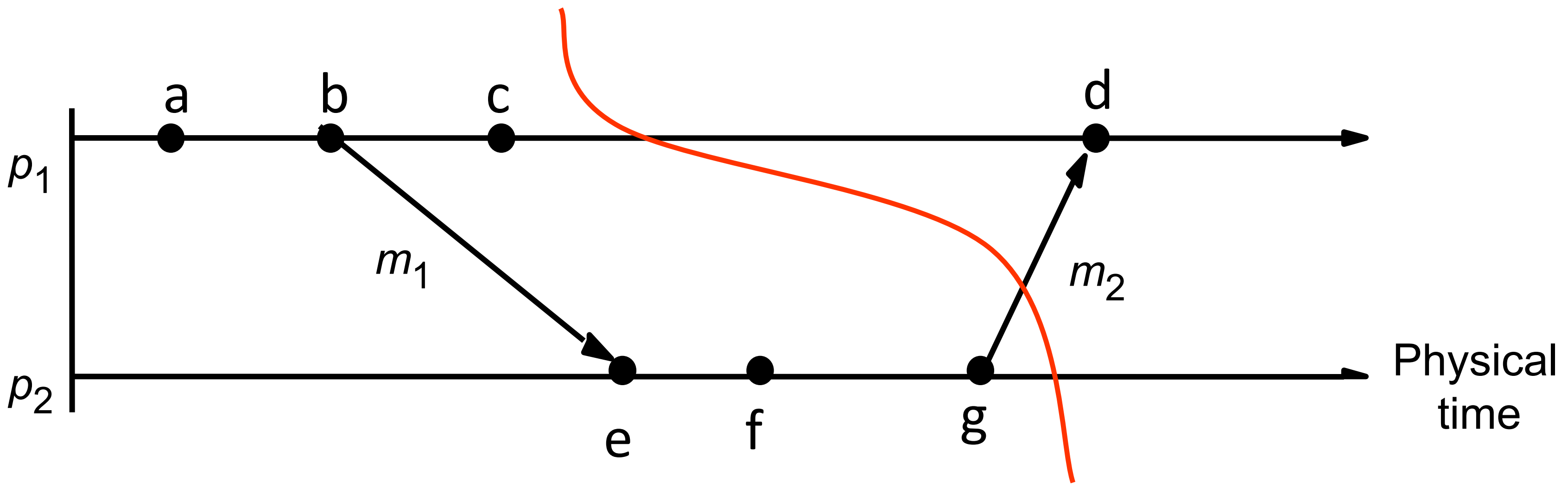
Linearization: < a, b, c, e, | f, g , d >

Example



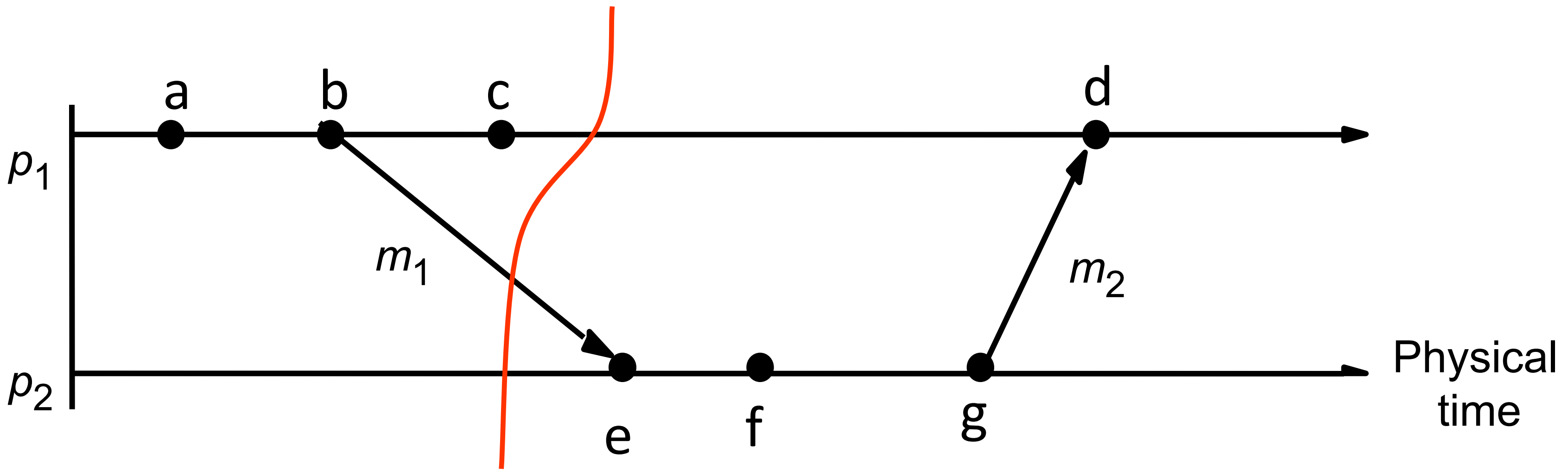
Linearization: < a, b, c, e, f, g , d >

Example



Linearization: < a, b, c, e, f, g, d >

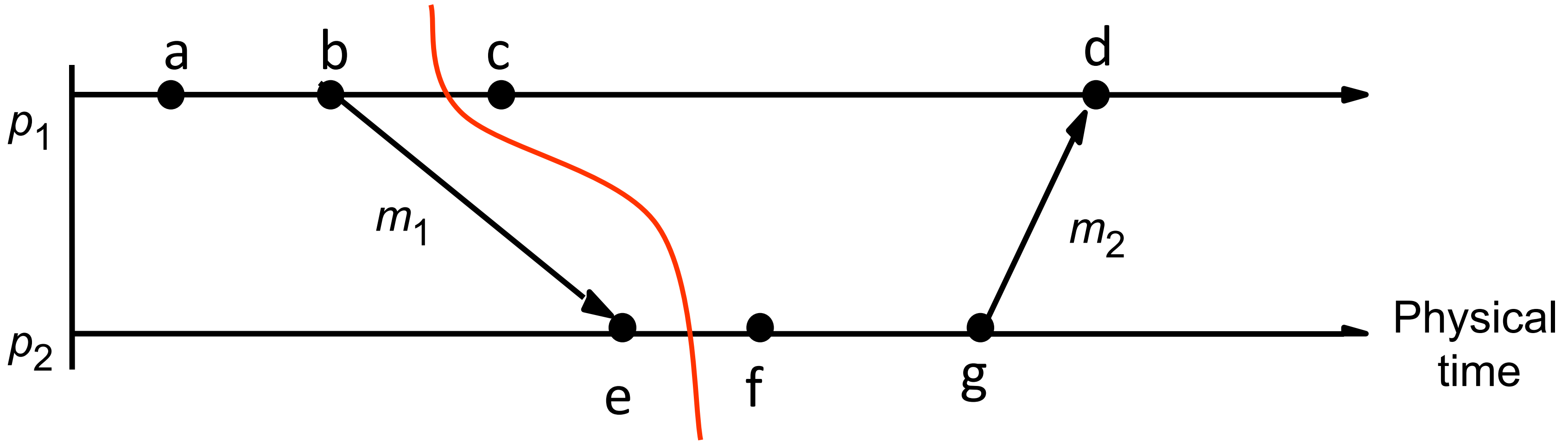
Example



Linearization: < a, b, c, e, f, g , d >

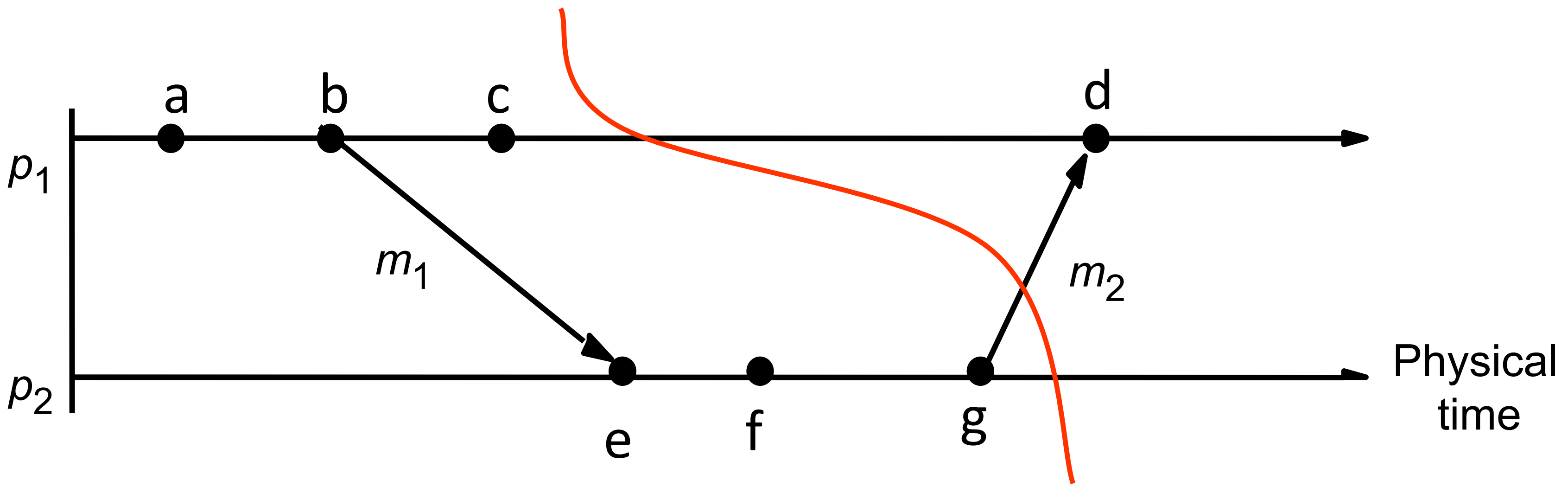
Linearization: < a, b, e, c, f, g , d >

Example



Linearization: < a, b, c, e, f, g , d >
Linearization: < a, b, e, c, f, g , d >

Example



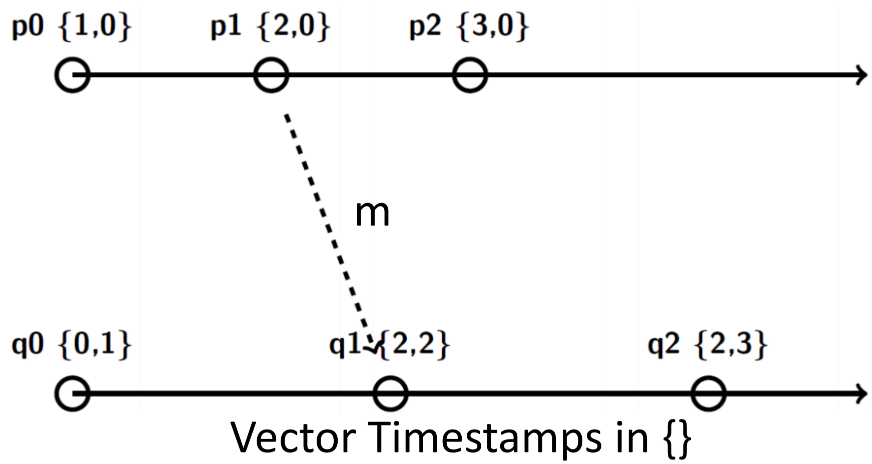
Linearization: < a, b, c, e, f, g | d >

Linearization: < a, b, e, c, f, g | d >

More notations and definitions

- A run is a total ordering of events in H that is consistent with each $\mathbf{h_i}$'s ordering.
- A linearization is a run consistent with happens-before ($\rightarrow$) relation in H.
- Linearizations pass through consistent global states.
- A global state $\mathbf{S_k}$ is reachable from global state $\mathbf{S_i}$, if there is a linearization that passes through $\mathbf{S_i}$ and then through $\mathbf{S_k}$.
- The distributed system evolves as a series of transitions between global states S_0 , S_1 , ….

State Transitions: Example



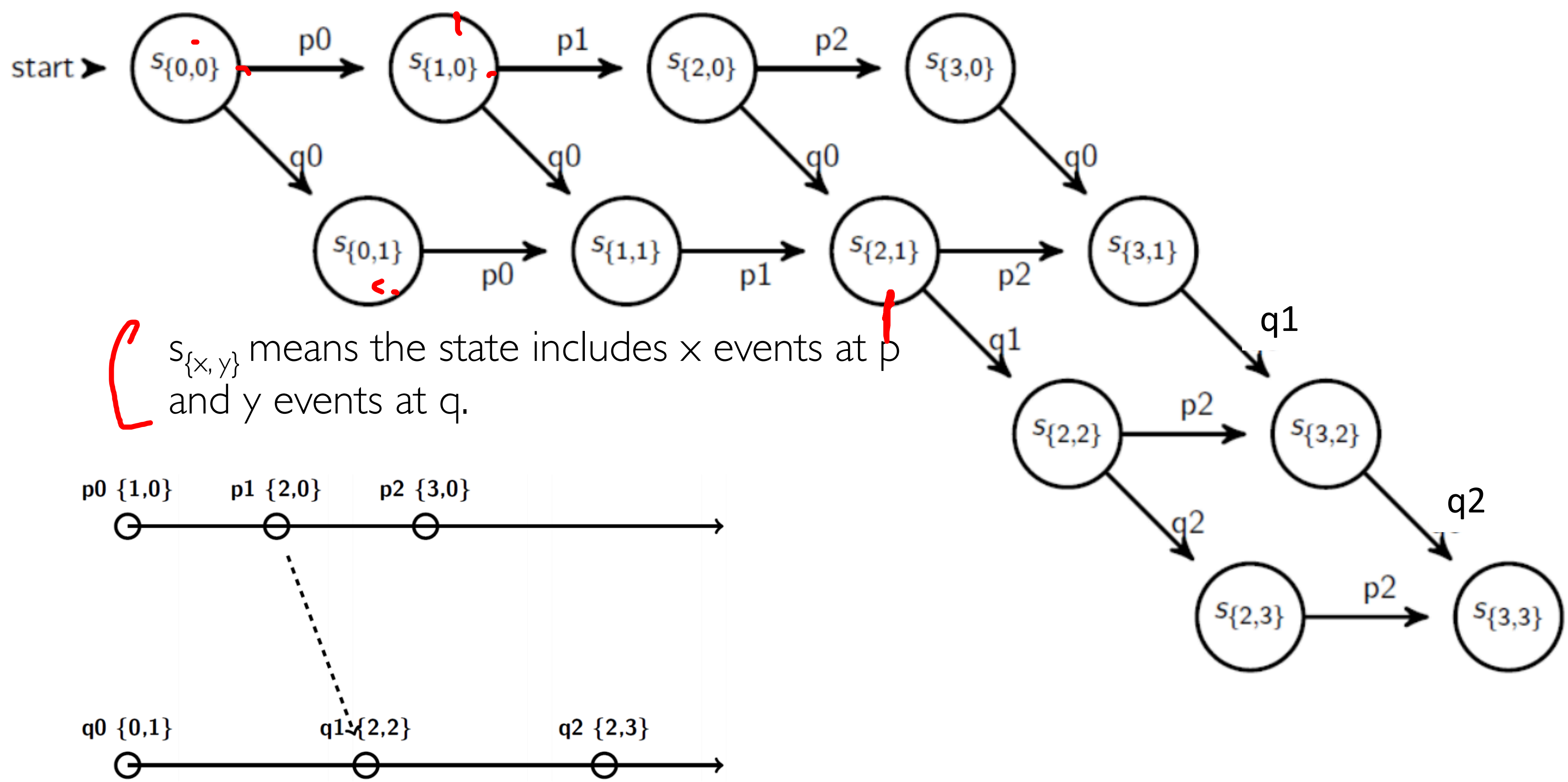
Many linearizations:

- < p0, p1, p2, q0, q1, q2>
- < p0, q0, p1, q1, p2, q2>
- <q0, p0, p1, q1, p2, q2 >
- <q0, p0, p1, p2, q1,q2 >
- ……

- Causal order:
  - p0 → p1 → p2
  - q0 → q1 → q2
  - p0 → p1 → q1 → q2
- Concurrent:
  - p0 || q0
  - p1 || q0
  - p2 || q0, p2 || q1, p2 || q2

State Transitions: Example

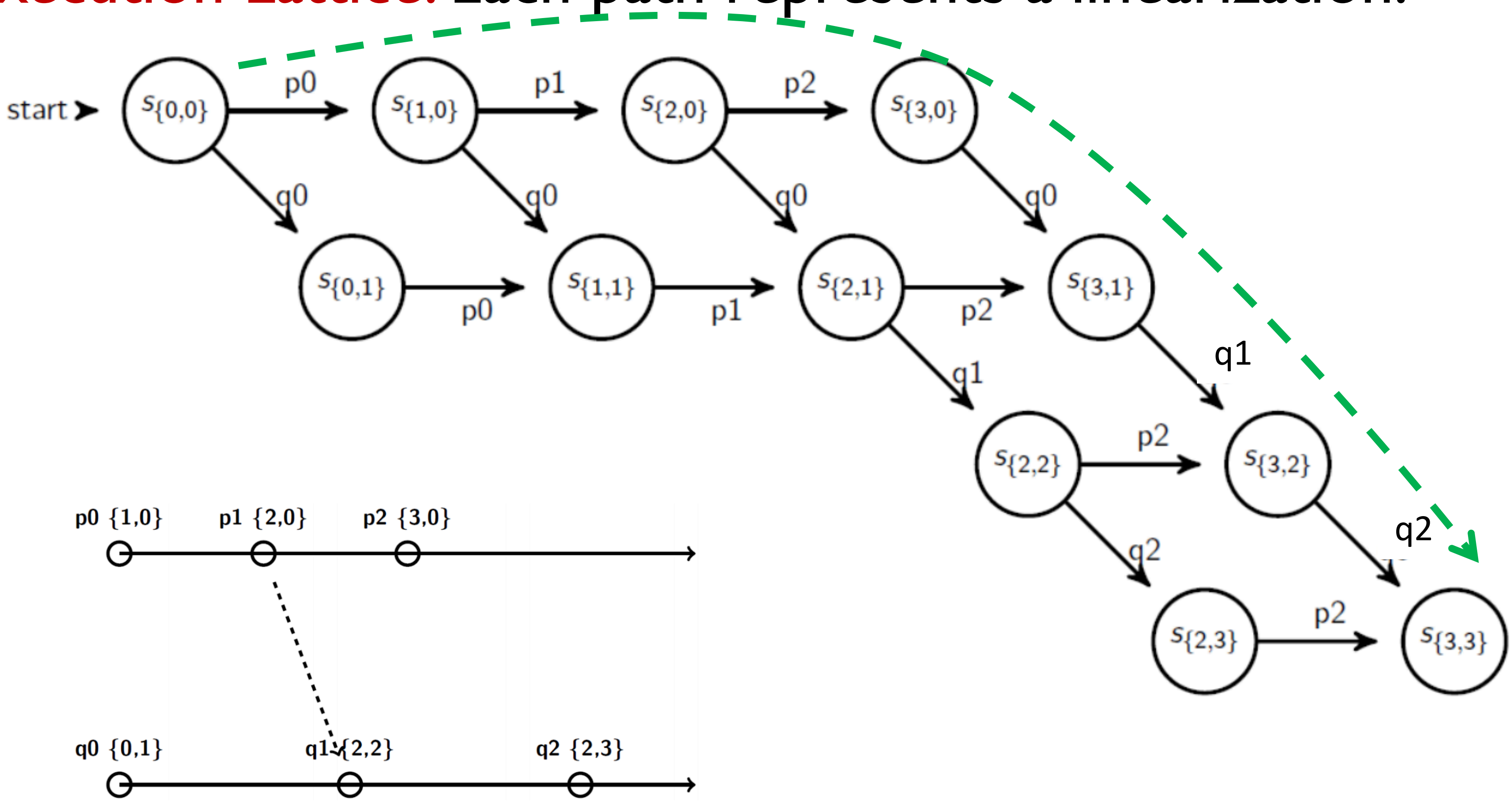
Execution Lattice. Each path represents a linearization.



$s_{\{x, y\}}$ means the state includes x events at p and y events at q.

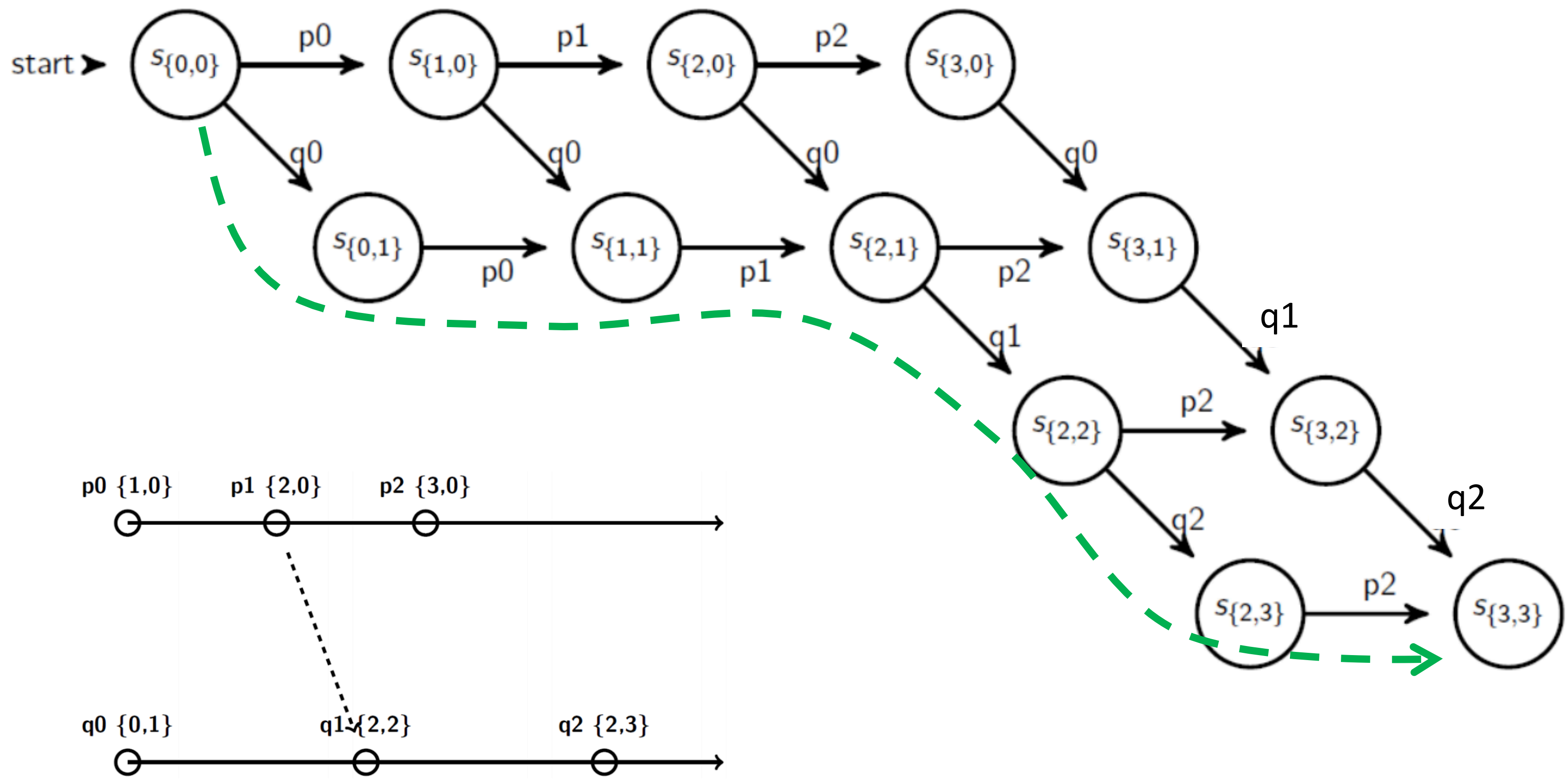
State Transitions: Example

Execution Lattice. Each path represents a linearization.



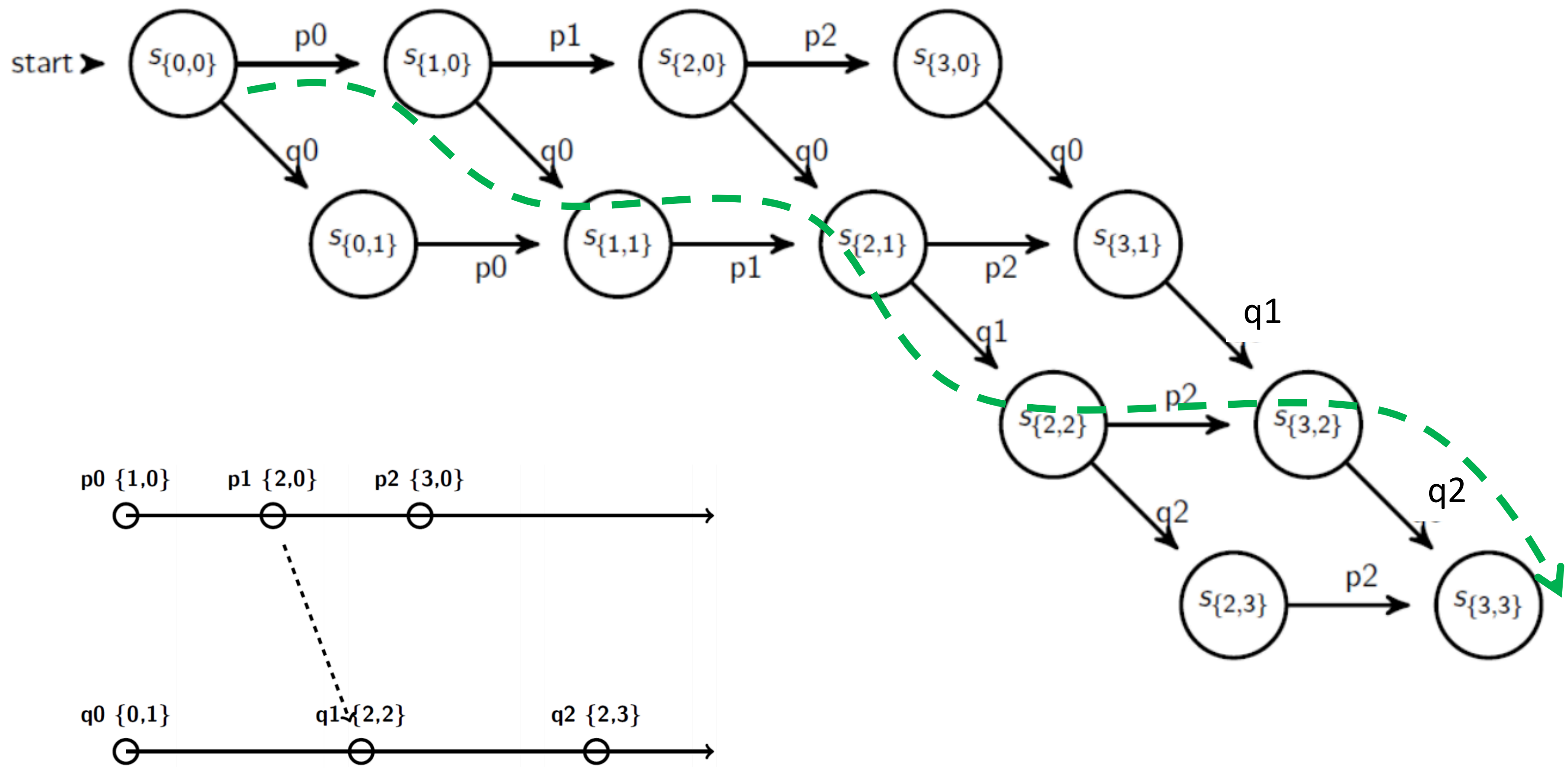
State Transitions: Example

Execution Lattice. Each path represents a linearization.



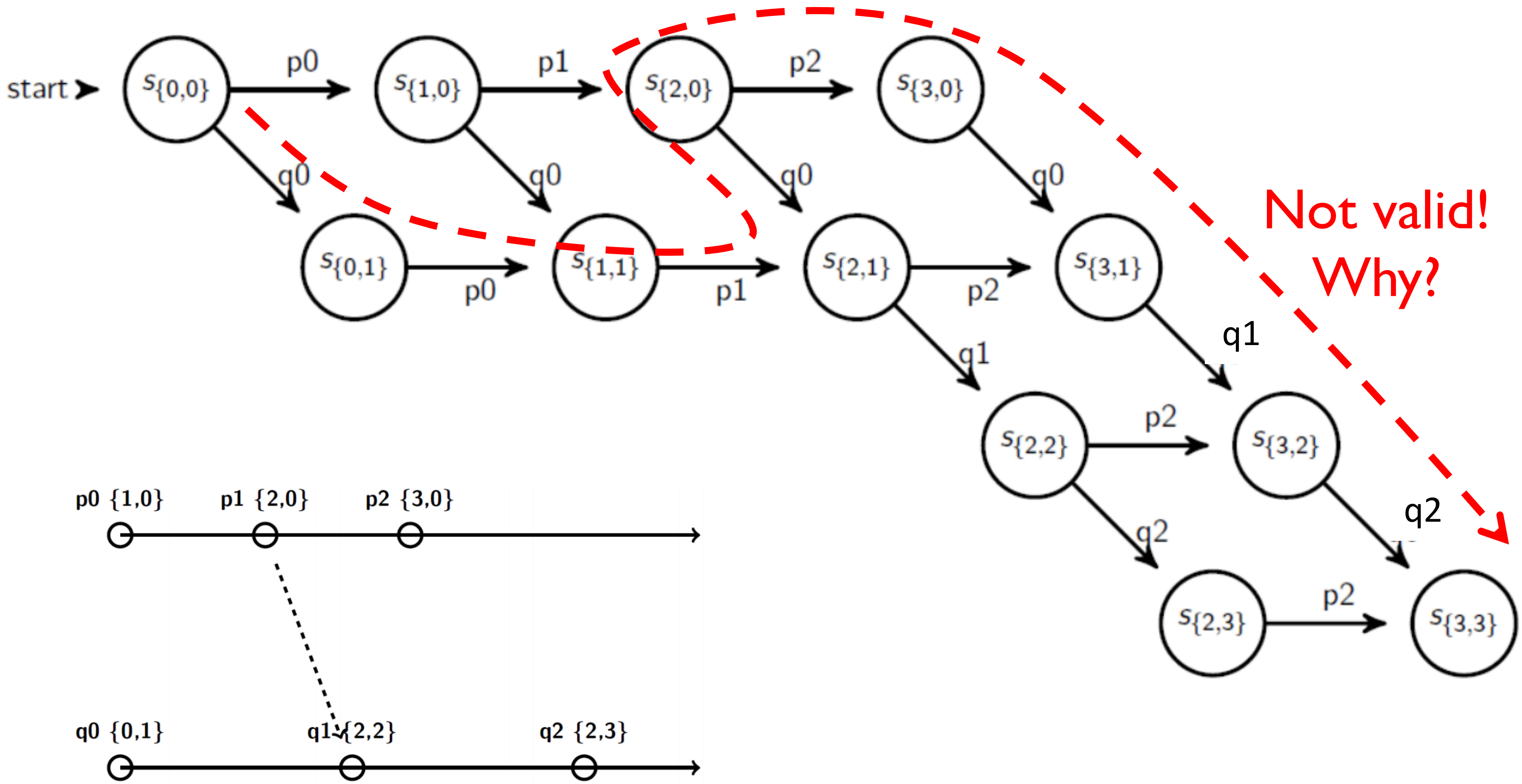
State Transitions: Example

Execution Lattice. Each path represents a linearization.

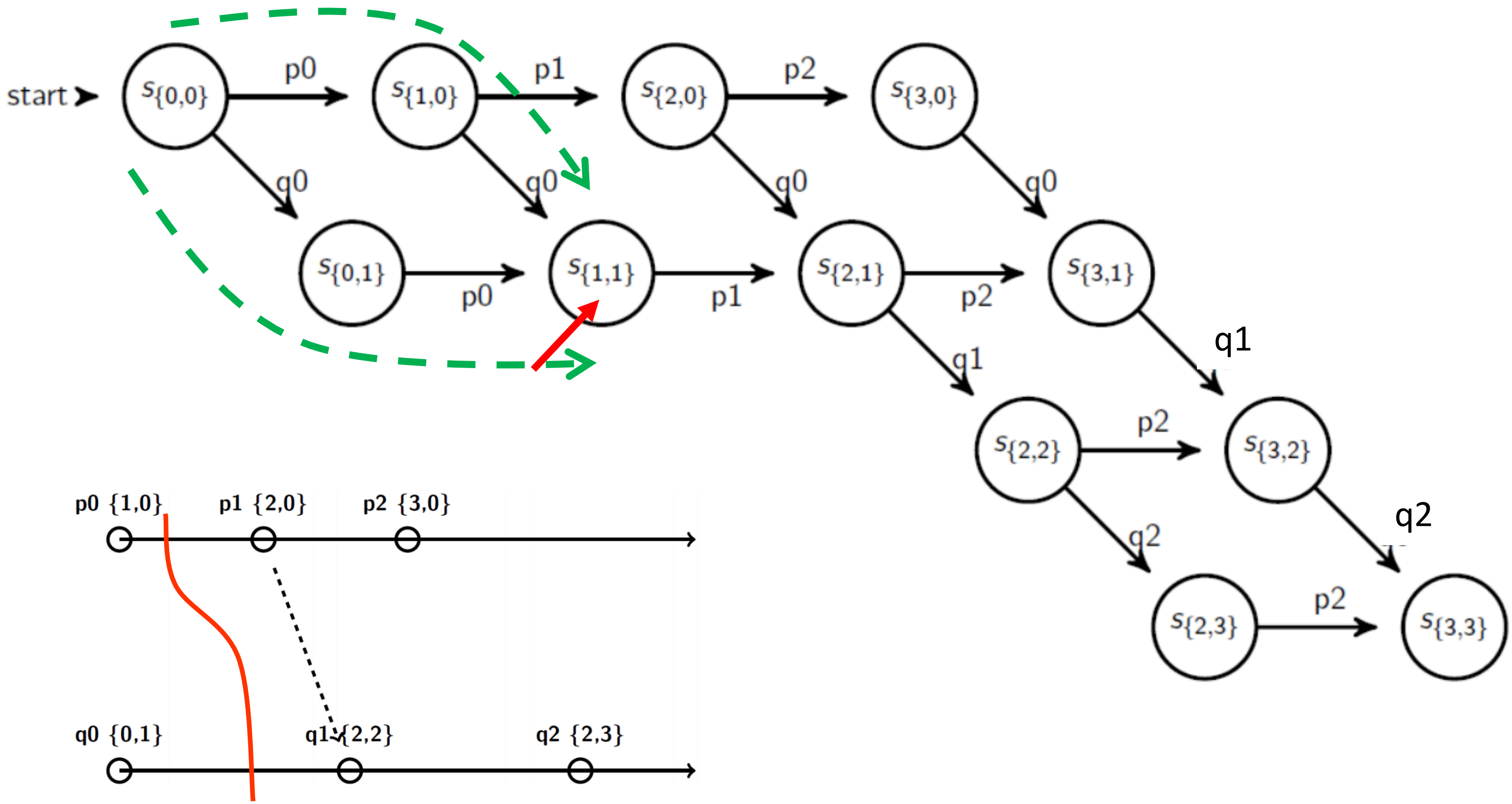


State Transitions: Example

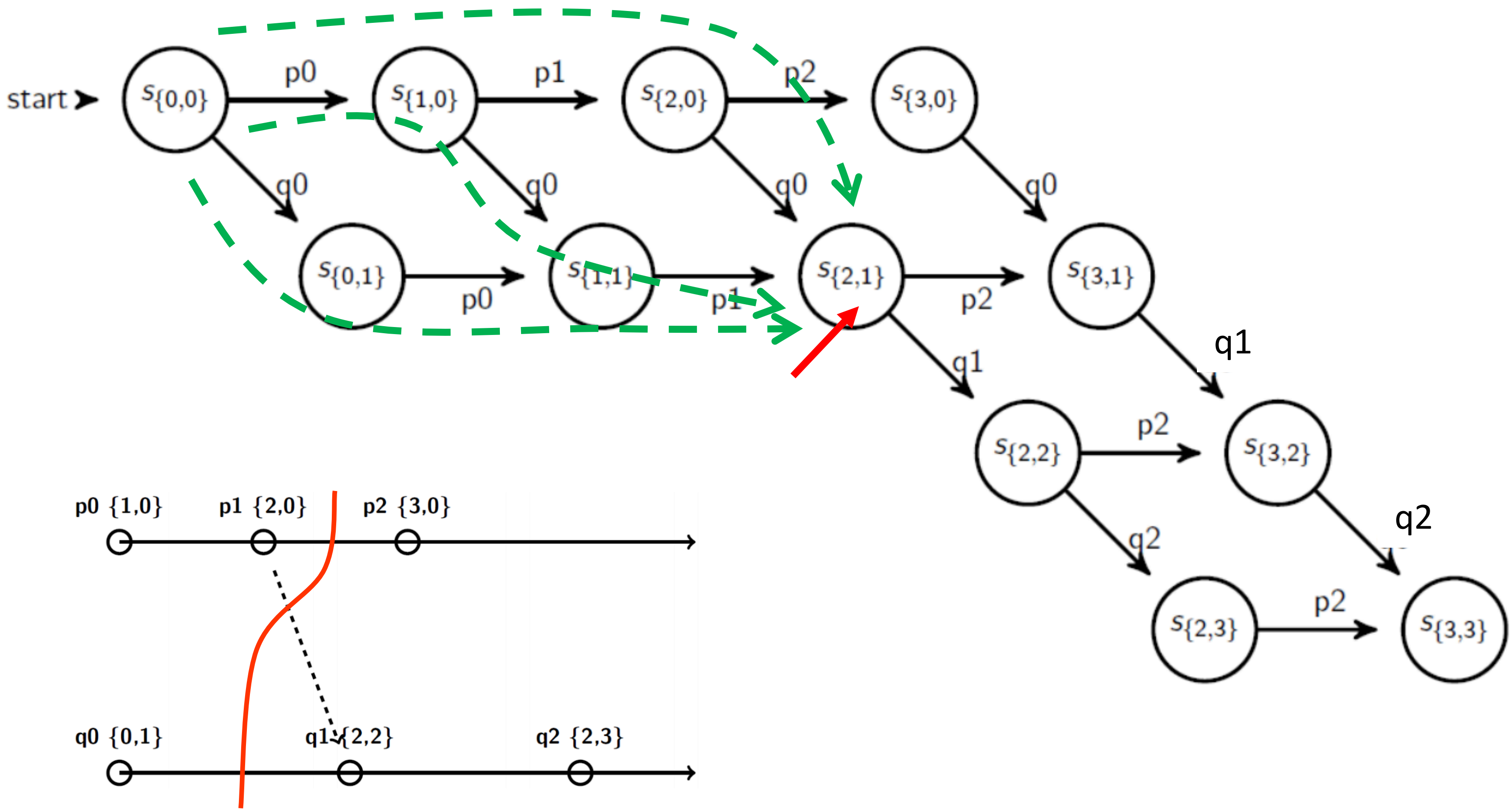
Execution Lattice. Each path represents a linearization.



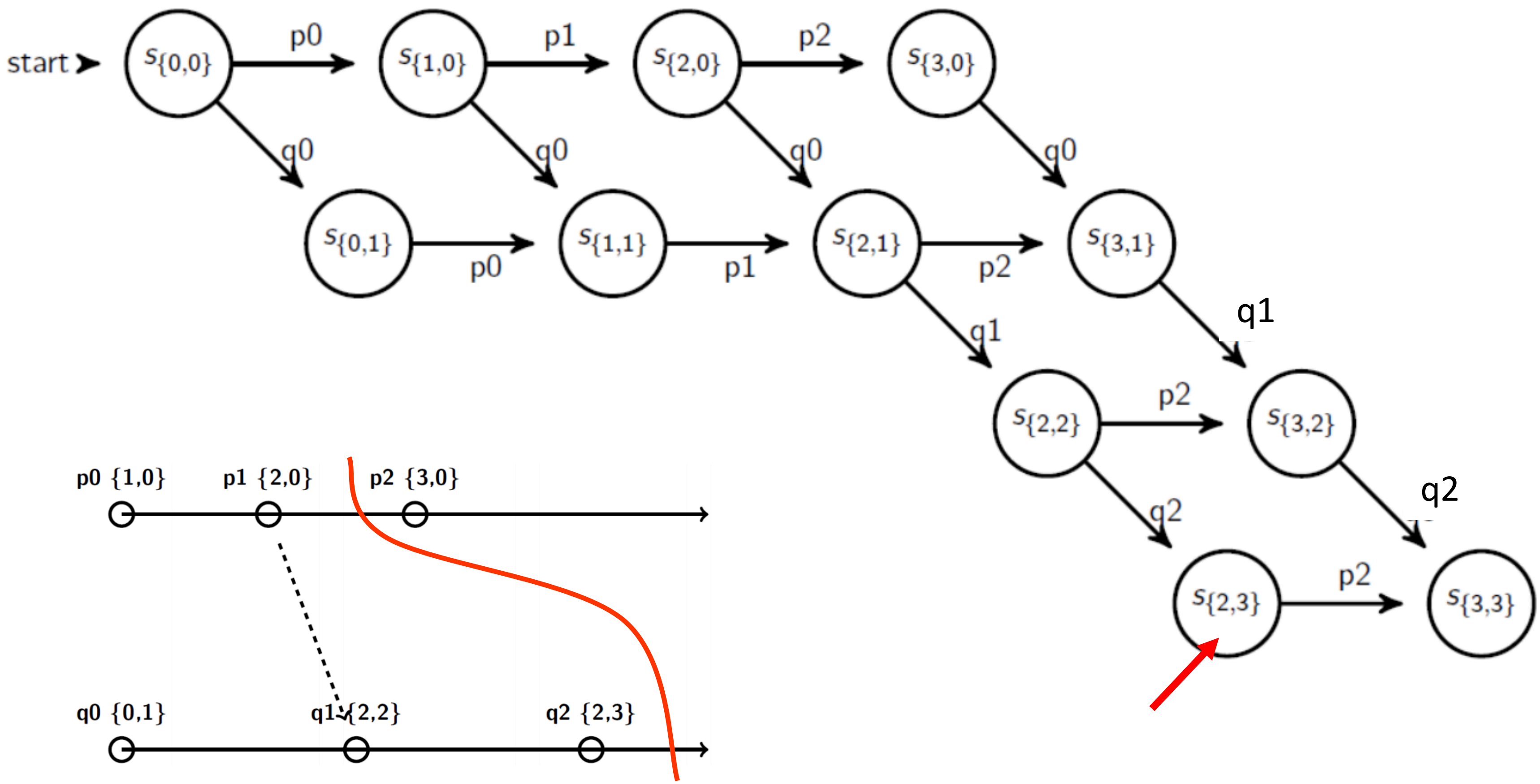
State Transitions: Example



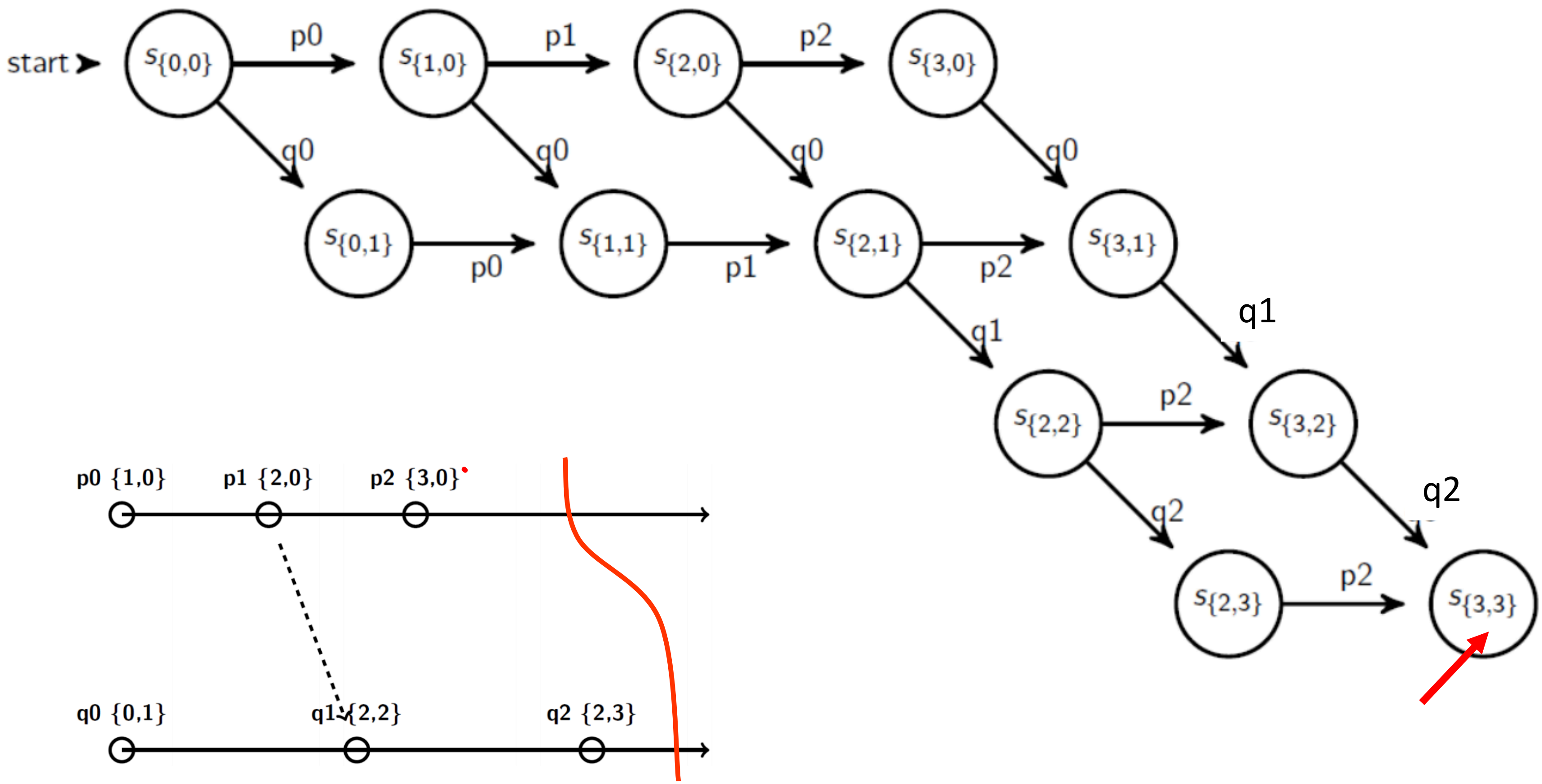
State Transitions: Example



State Transitions: Example



State Transitions: Example



More notations and definitions

- A run is a total ordering of events in H that is consistent with each $\mathbf{h_i}$'s ordering.
- A linearization is a run consistent with happens-before ($\rightarrow$) relation in H.
- Linearizations pass through consistent global states.
- A global state $\mathbf{S_k}$ is reachable from global state $\mathbf{S_i}$, if there is a linearization that passes through $\mathbf{S_i}$ and then through $\mathbf{S_k}$.
- The distributed system evolves as a series of transitions between global states S_0 , S_1 , ….

Global State Predicates

- A global-state-predicate is a property that is *true* or *false* for a global state.
  - Is there a deadlock?
  - Has the distributed algorithm terminated?
- Two ways of reasoning about predicates (or system properties) as global state gets transformed by events.
  - Liveness
  - Safety

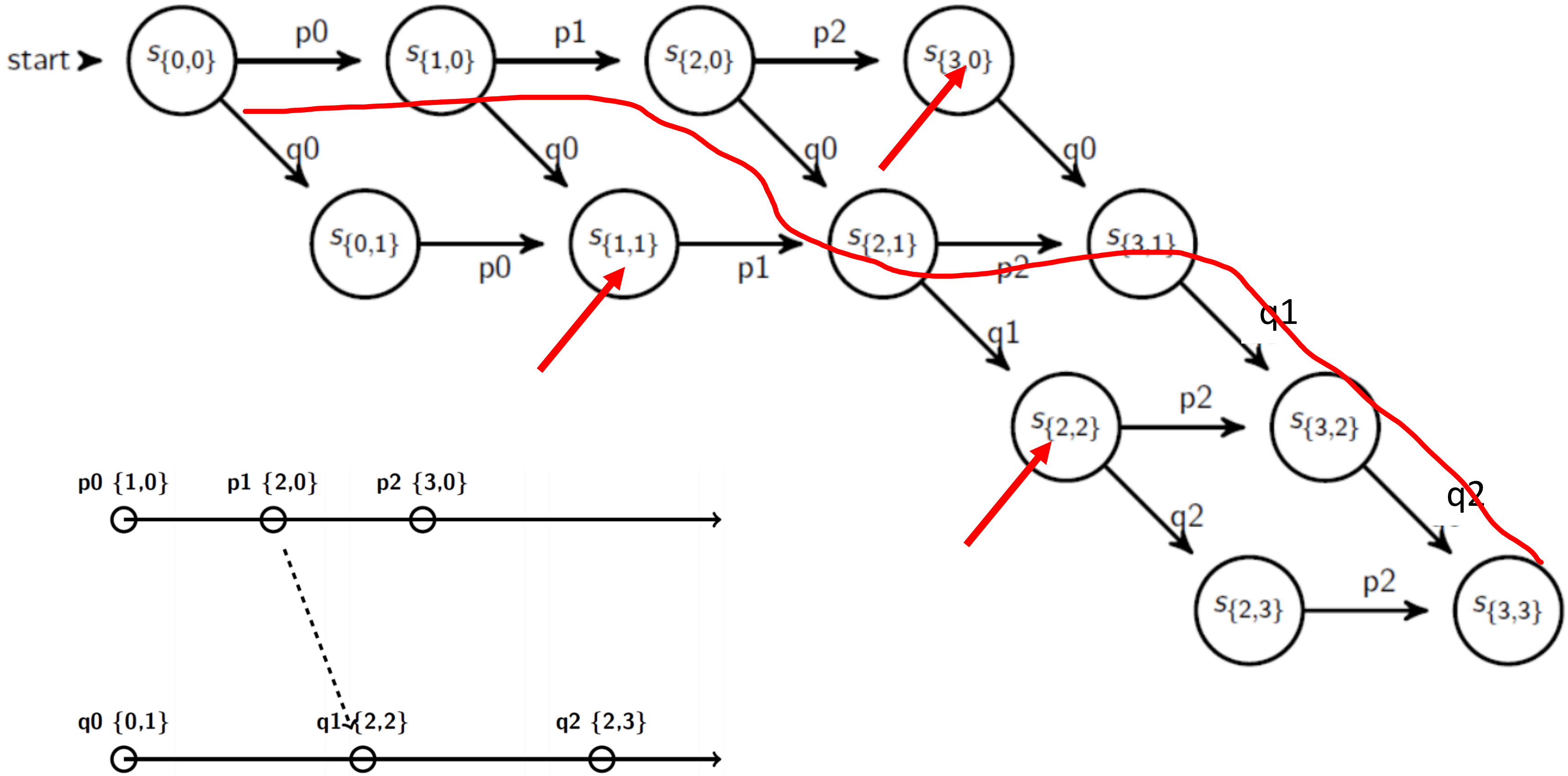
Liveness

- **Liveness = guarantee that something good will happen, eventually**
- **Examples:**
  - A distributed computation will terminate.
  - "Completeness" in failure detectors: the failure will be detected.
  - All processes will eventually decide on a value.
- A global state S_0 satisfies a **liveness** property P iff:
  - For all linearizations starting from S_0, P is true for some state S_L reachable from S_0.
  - $liveness(P(S_0)) \equiv \forall L \in$ linearizations from S_0, L passes through a S_L & $P(S_L)$ = true

Liveness Example

If predicate is true only in the marked states, does it satisfy liveness?

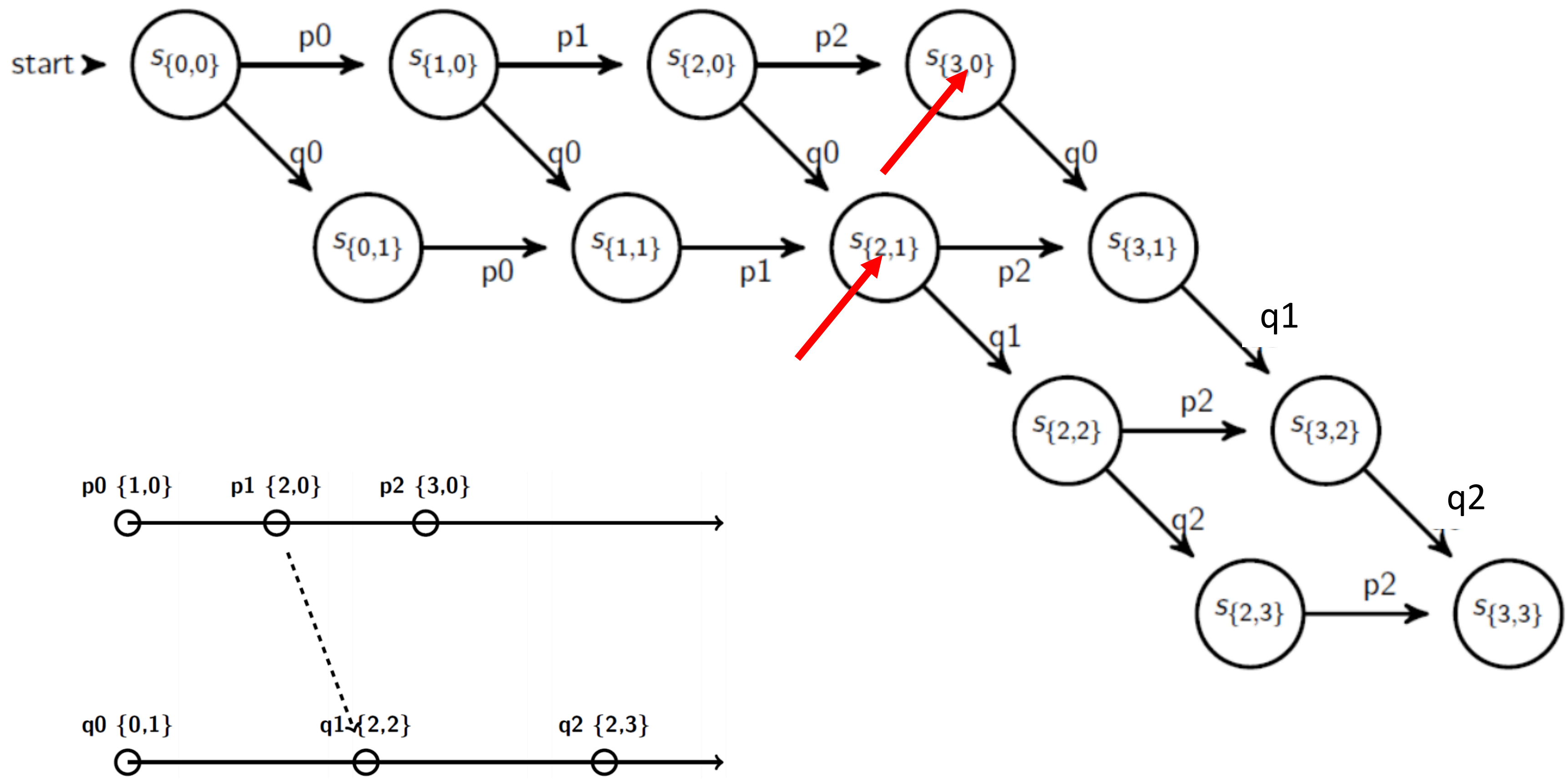
No



Liveness Example

If predicate is true only in the marked states, does it satisfy liveness?

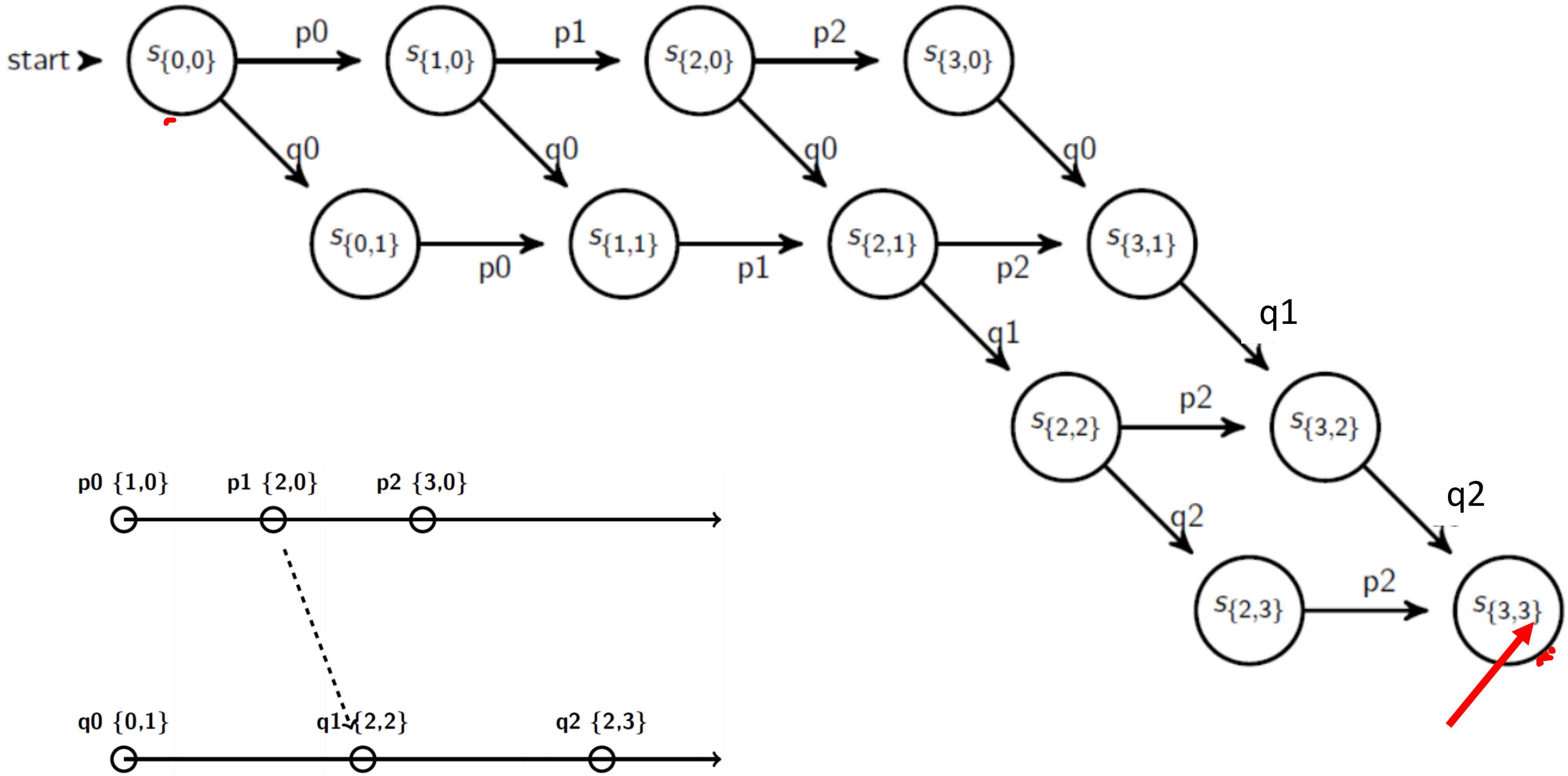
Yes



Liveness Example

If predicate is true only in the marked states, does it satisfy liveness?

Yes



Liveness

- **Liveness = guarantee that something good will happen, eventually**

- **Examples:**
  - A distributed computation will terminate.
  - "Completeness" in failure detectors: the failure will be detected.
  - All processes will eventually decide on a value.

- A global state S_0 satisfies a **liveness** property P iff:
  - liveness(P(S_0)) $\equiv$ $\forall L \in$ linearizations from S_0, L passes through a S_L & P(S_L) = true
  - For any linearization starting from S_0, P is true for some state S_L reachable from S_0.

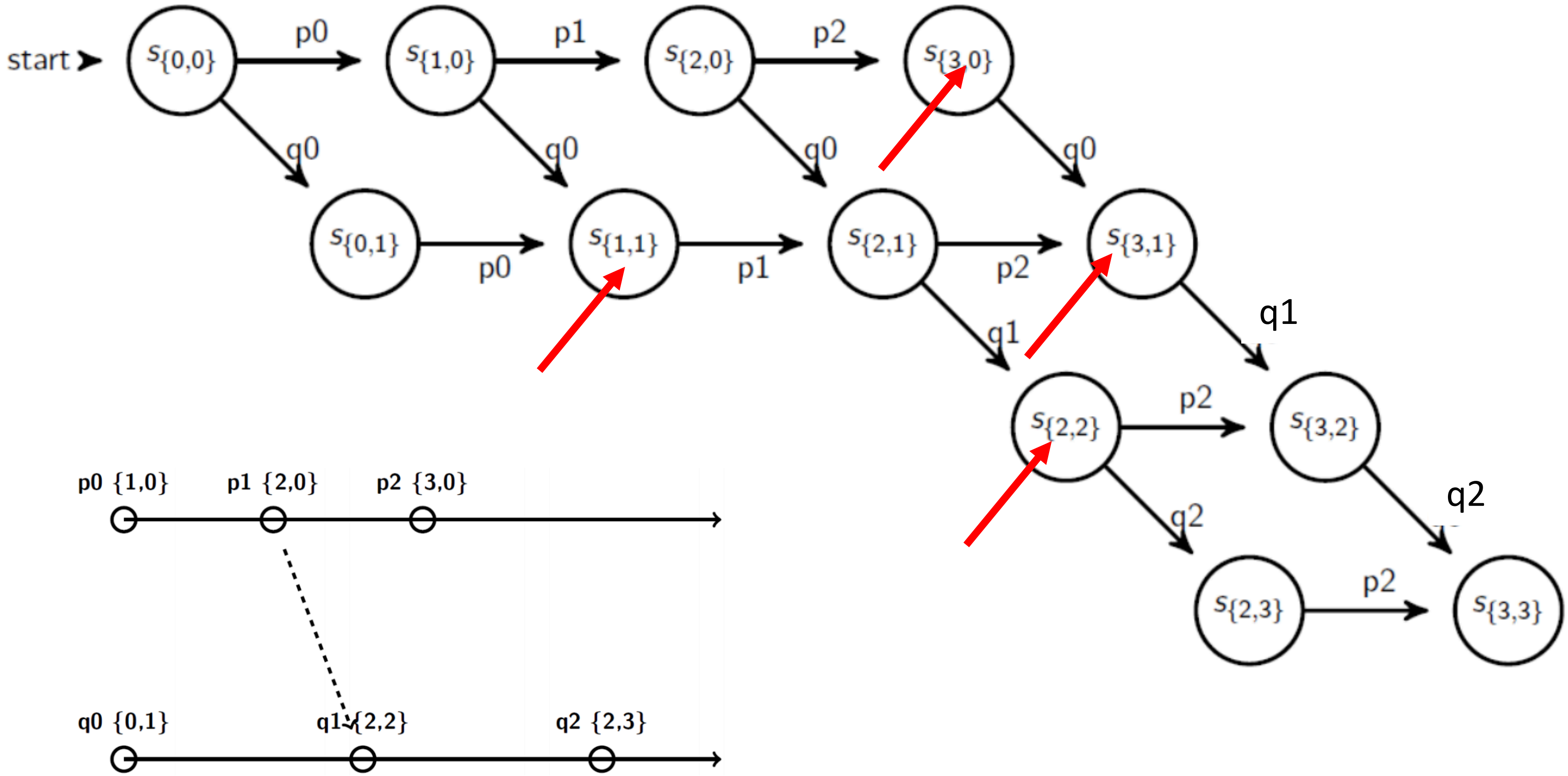
Safety

- **Safety = guarantee that something bad will never happen.**
- **Examples:**
  - There is no deadlock in a distributed transaction system.
  - "Accuracy" in failure detectors: an alive process is not detected as failed.
  - No two processes decide on different values.
- A global state S_0 satisfies a **safety** property P iff:
  - For all states S reachable from S_0, P(S) is true.
  - $safety(P(S_0)) \equiv \forall S$ reachable from S_0, $P(S) = true$.

Safety Example

If predicate is true only in the marked states, does it satisfy safety?

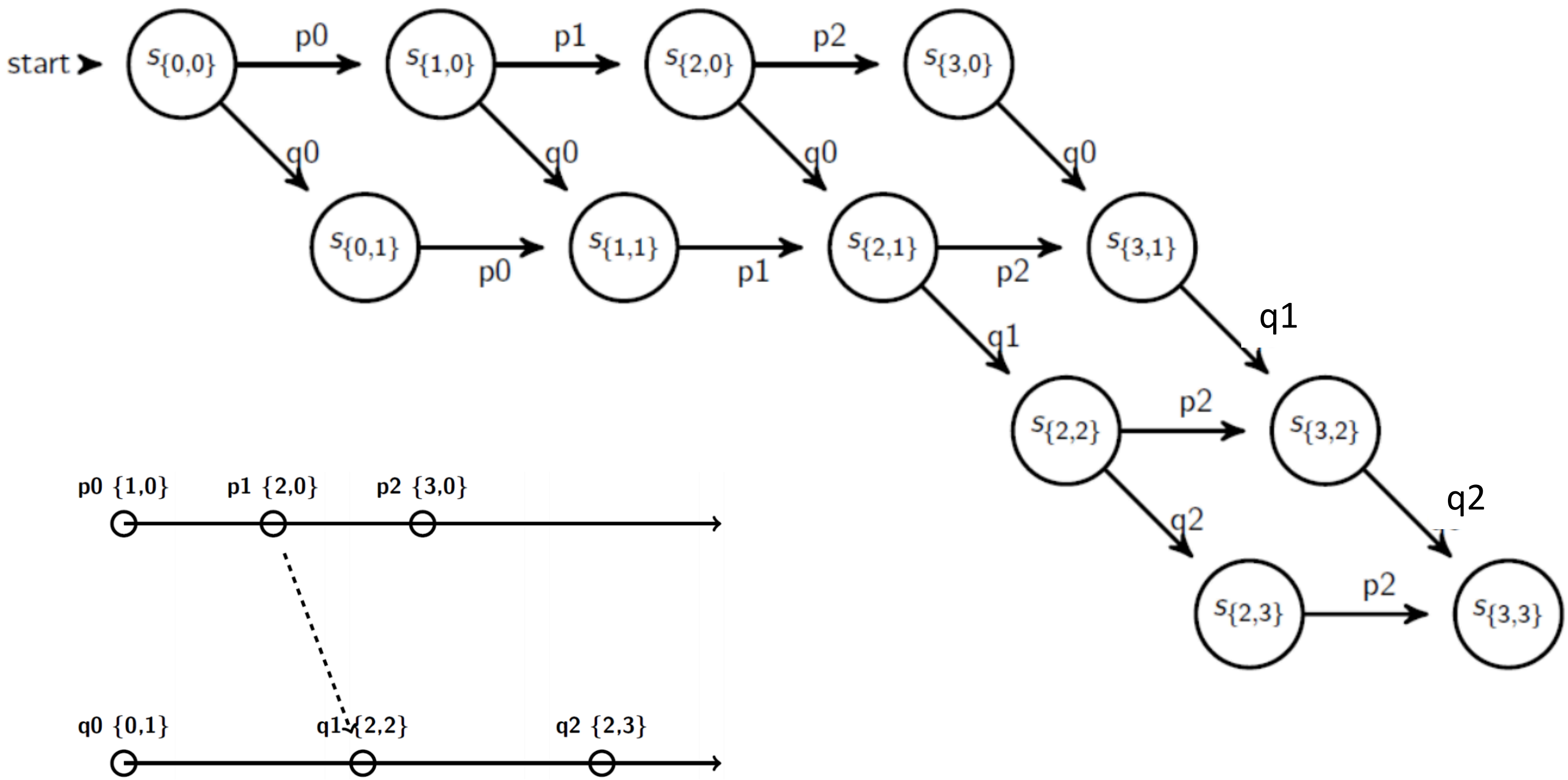
No



Safety Example

If predicate is true only in the **unmarked** states, does it satisfy safety?

Yes

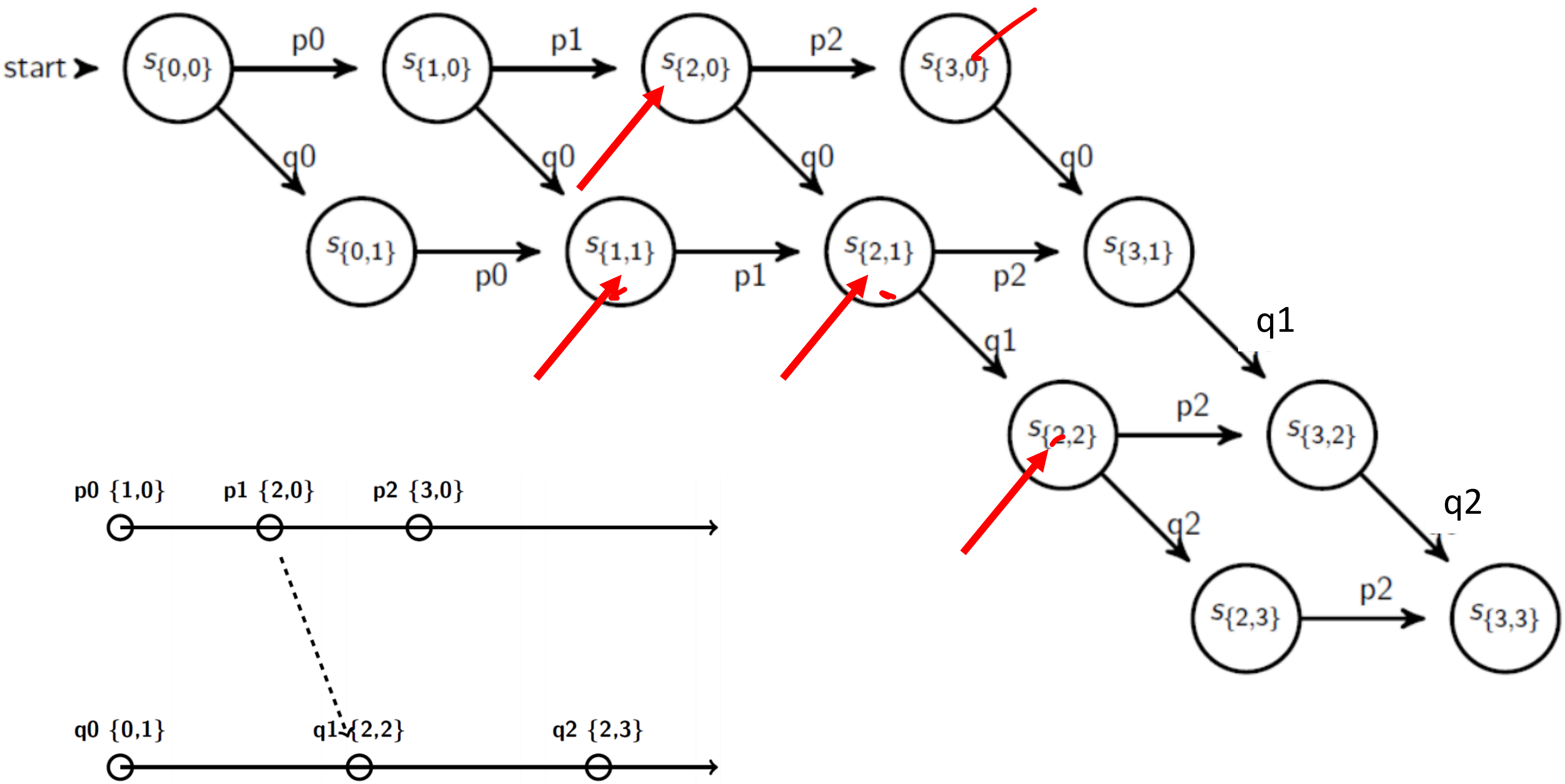


Safety

- **Safety = guarantee that something bad will never happen.**
- **Examples:**
  - There is no deadlock in a distributed transaction system.
  - "Accuracy" in failure detectors: an alive process is not detected as failed.
  - No two processes decide on different values.
- A global state S_0 satisfies a **safety** property P iff:
  - $\text{safety}(P(S_0)) \equiv \forall S$ reachable from S_0, $P(S) = \text{true}$.
  - For all states S reachable from S_0, P(S) is true.

Liveness Example

Technically satisfies liveness, but difficult to capture or reason about.



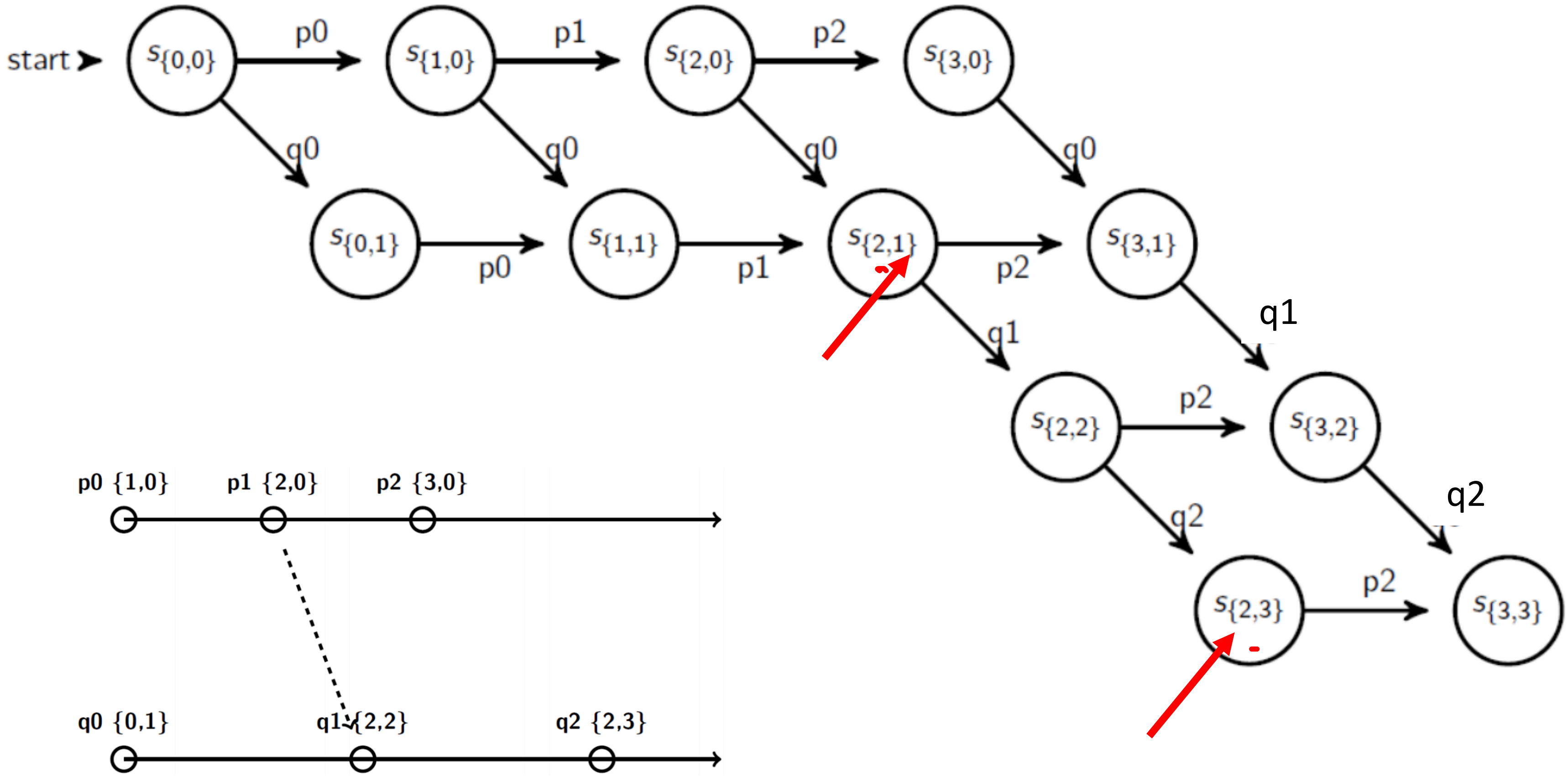
Stable Global Predicates

- once true, stays true forever afterwards (for stable liveness)

Stable Global Predicates

If predicate is true only in the marked states, is it true in a stable way?

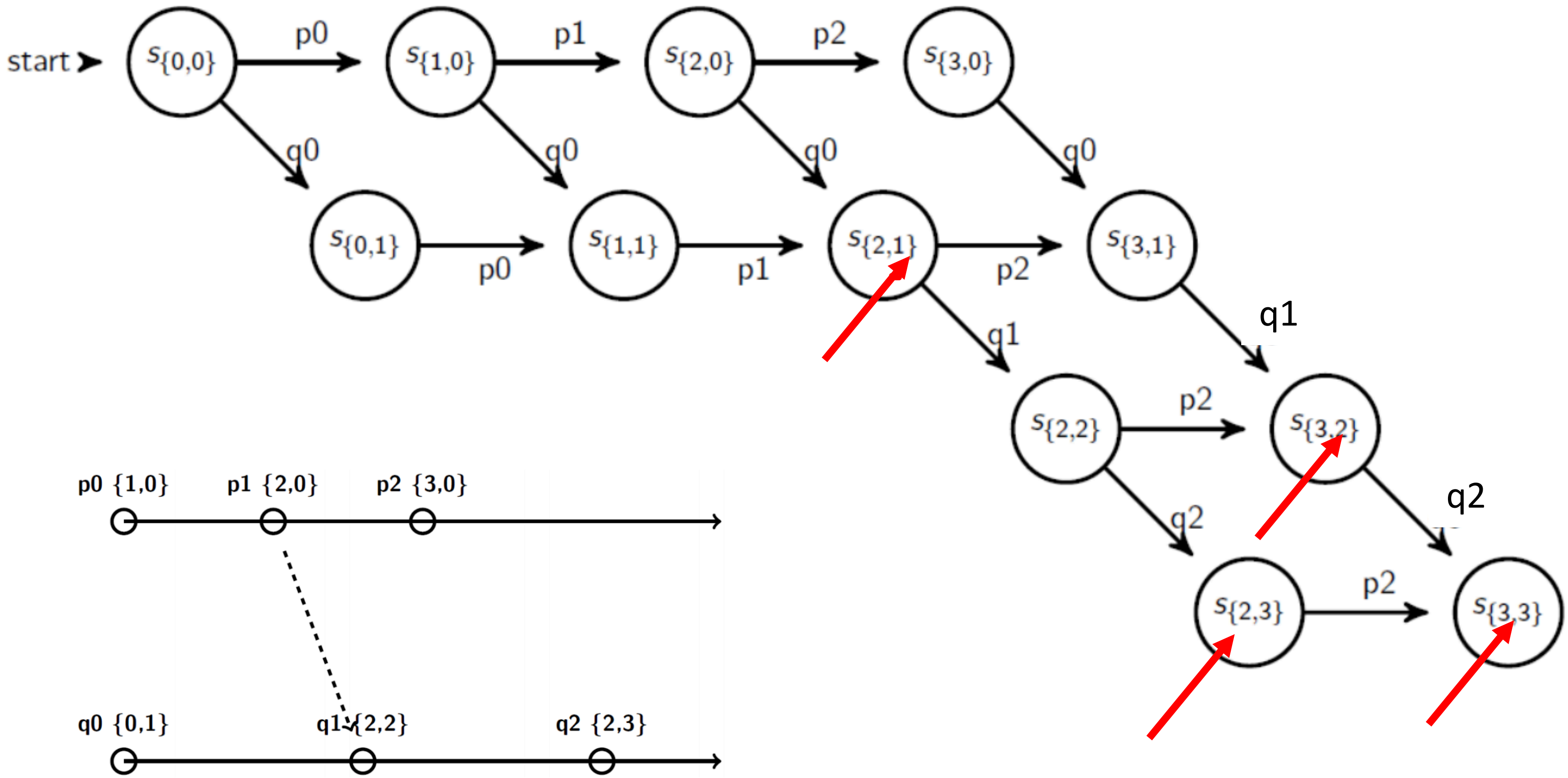
No



Stable Global Predicates

If predicate is true only in the marked states, is it true in a stable way?

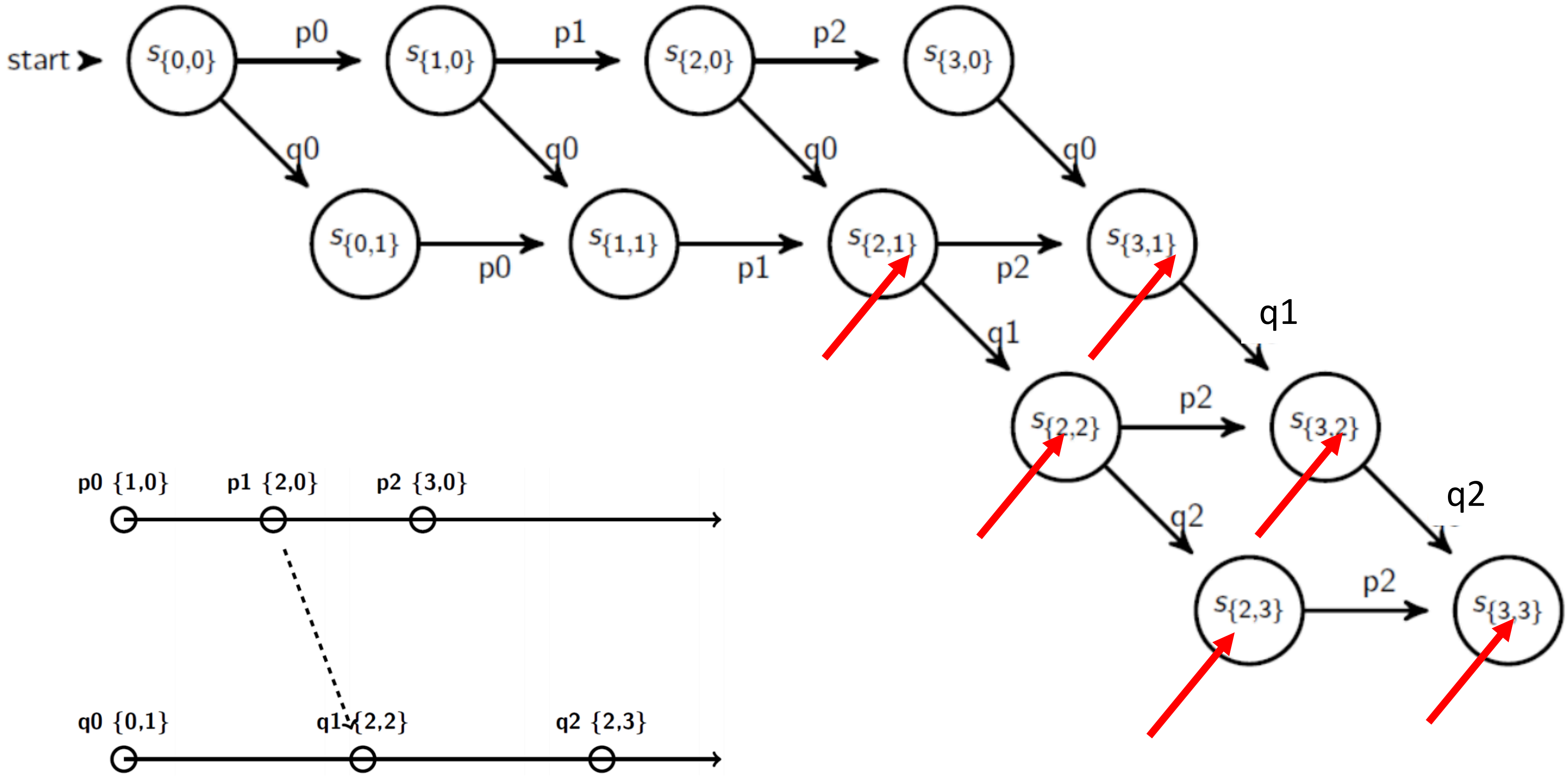
No



Stable Global Predicates

If predicate is true only in the marked states, is it true in a stable way?

Yes



Stable Global Predicates

- once true for a state S, stays true for all states reachable from S (for stable liveness)
- once false for a state S, stays false for all states reachable from S (for stable non-safety)
- Stable liveness examples (once true, always true)
  - Computation has terminated.
- Stable non-safety examples (once false, always false)
  - There is no deadlock.
  - An object is not orphaned.
- *All stable global properties can be detected using the Chandy-Lamport algorithm.*

Global Snapshot Summary

- The ability to calculate global snapshots in a distributed system is very important.
- But don't want to interrupt running distributed application.
- Chandy-Lamport algorithm calculates global snapshot.
- Obeys causality (creates a consistent cut).
- Can be used to detect global properties.
- Safety vs. Liveness.

Today's agenda

- Global State (contd.)
  - Chapter 14.5
- Multicast
  - Chapter 15.4
  - **Goal:** reason about desirable properties for message delivery among a group of processes.

Communication modes

- **Unicast**
  - Messages are sent from exactly one process to one process.
- **Broadcast**
  - Messages are sent from exactly one process to all processes on the network.
- **Multicast**
  - Messages broadcast within a group of processes.
  - A multicast message is sent from any one process to a group of processes on the network.

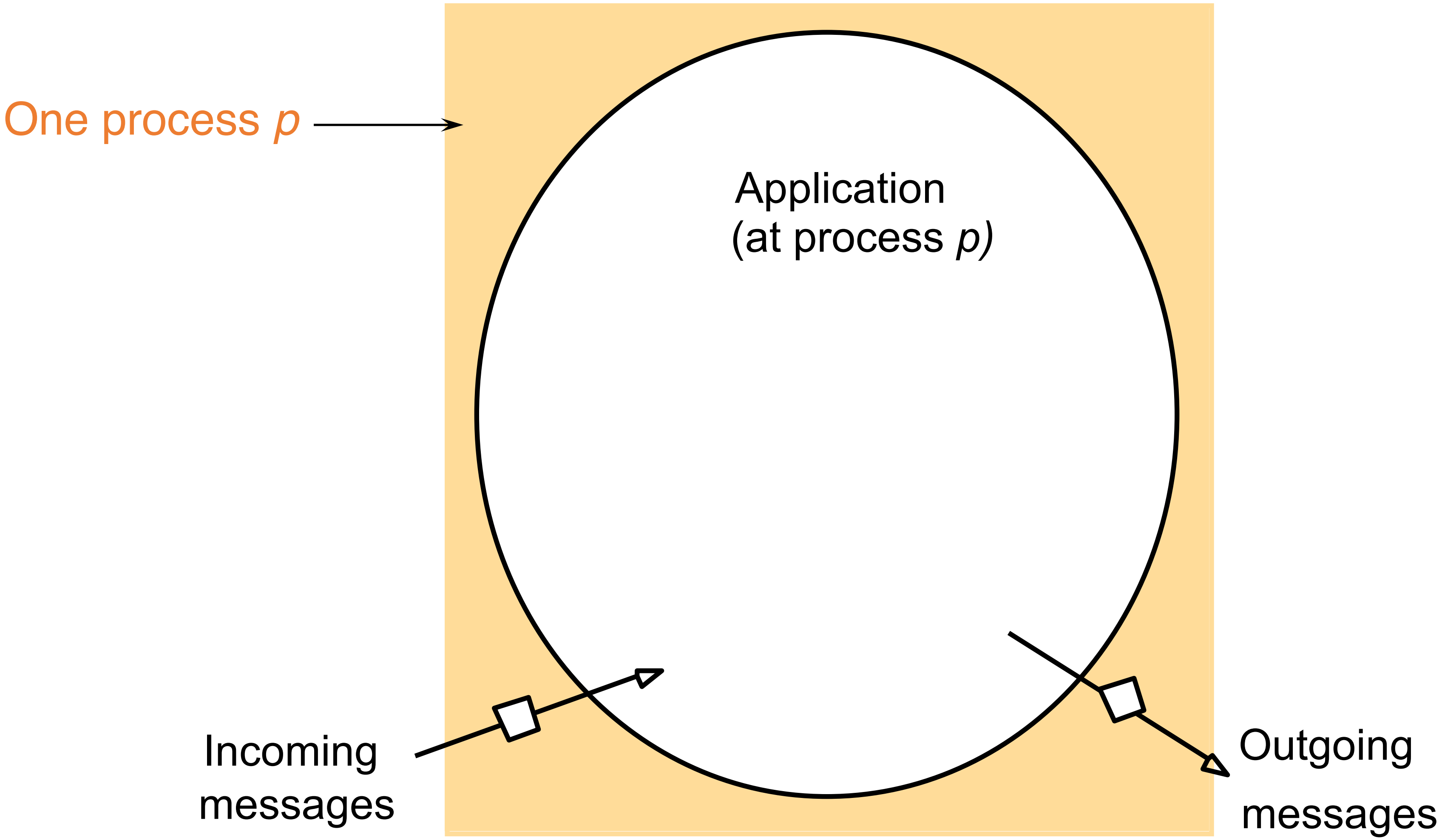
Where is multicast used?

- Distributed storage
  - Write to an object are multicast across replica servers.
  - Membership information (e.g., heartbeats) is multicast across all servers in cluster.
- Online scoreboards (ESPN, French Open, FIFA World Cup)
  - Multicast to group of clients interested in the scores.
- Stock Exchanges
  - Group is the set of broker computers.
- ……

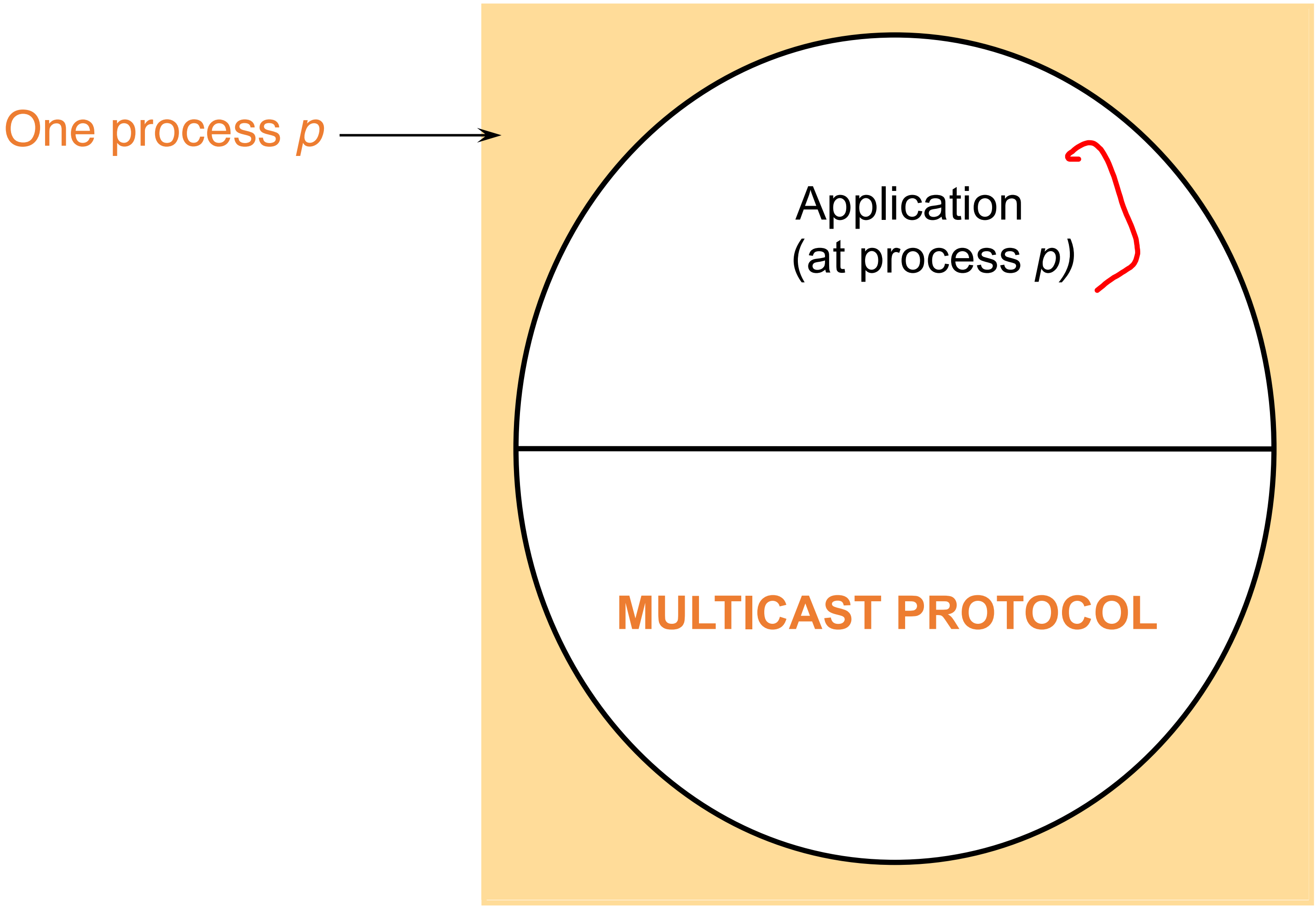
Communication modes

- **Unicast**
  - Messages are sent from exactly one process to one process.
    - *Best effort:* if a message is delivered it would be intact; no reliability guarantees.
    - *Reliable:* guarantees delivery of messages.
    - *In order:* messages will be delivered in the same order that they are sent.
- **Broadcast**
  - Messages are sent from exactly one process to all processes on the network.
- **Multicast**
  - Messages broadcast within a group of processes.
  - A multicast message is sent from any one process to the group of processes on the network.
  - *How do we define (and achieve) reliable or ordered multicast?*

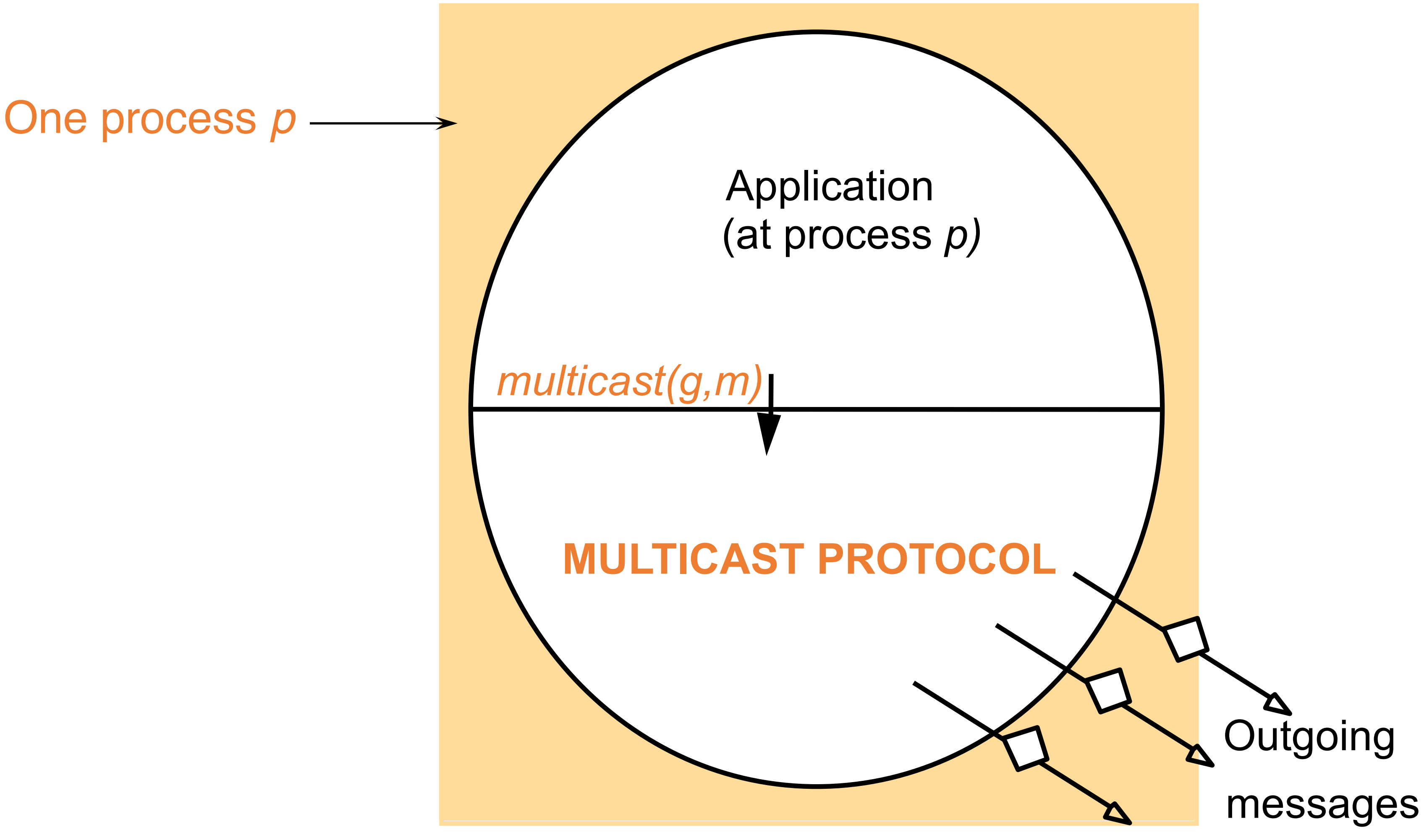
What we are designing in this class?



What we are designing in this class?

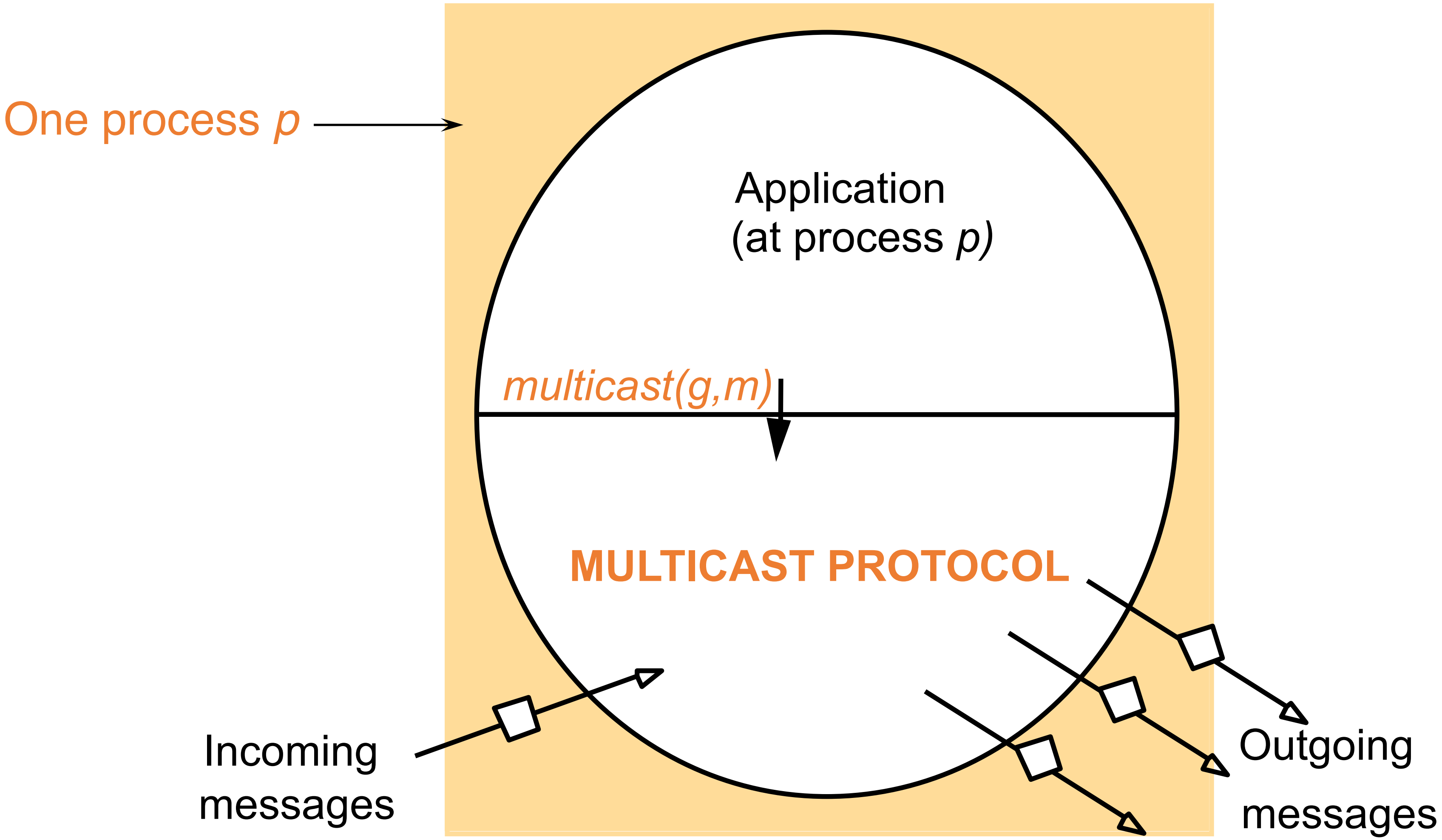


What we are designing in this class?



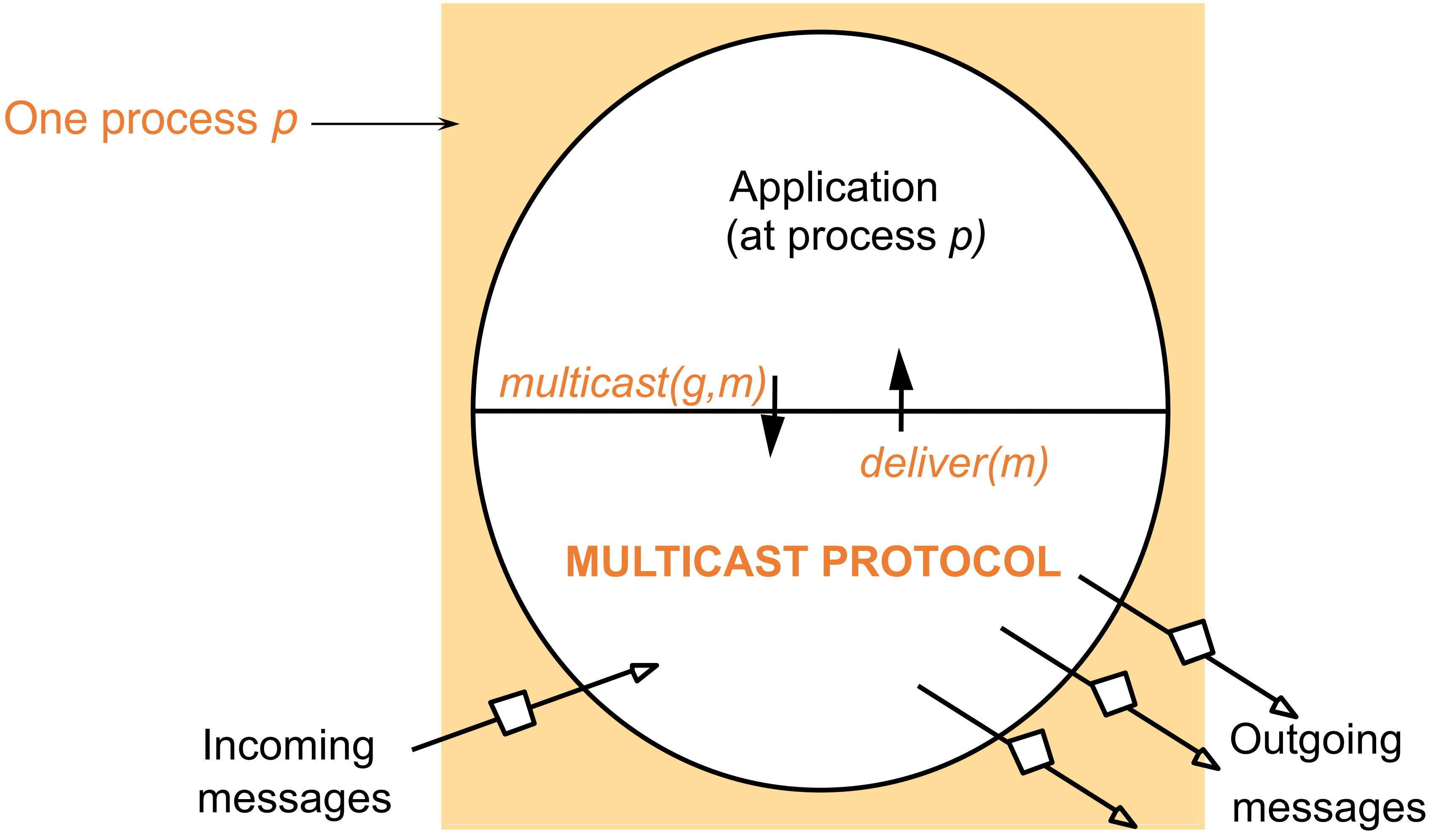
'g' is a multicast group that also includes the process 'p'.

What we are designing in this class?



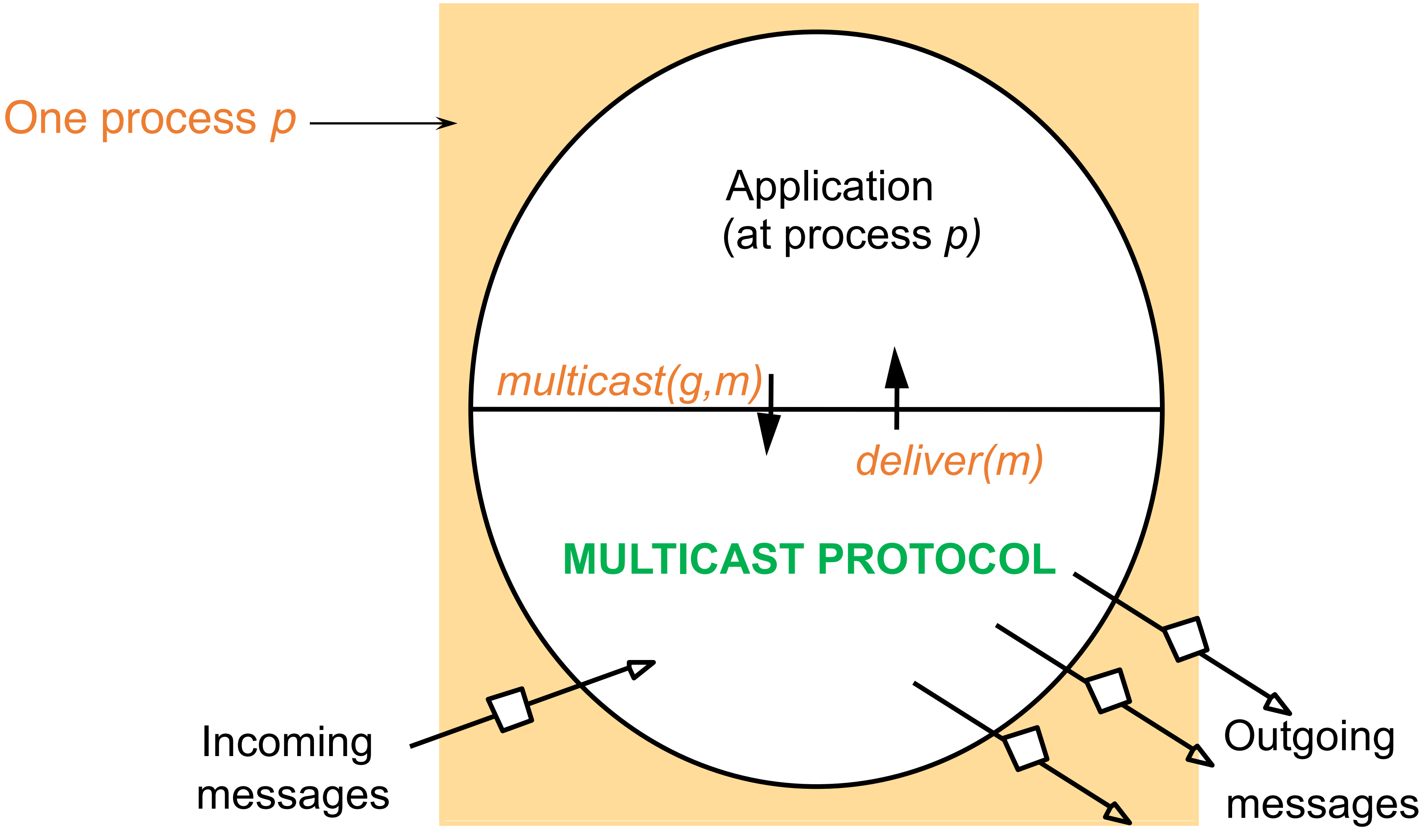
'g' is a multicast group that also includes the process 'p'.

What we are designing in this class?



‘g’ is a multicast group that also includes the process ‘p’.

What we are designing in this class?



'g' is a multicast group that also includes the process 'p'.

Basic Multicast (B-Multicast)

- Straightforward way to implement B-multicast:
  - use a reliable one-to-one send (unicast) operation:

    B-multicast(group g, message m):

    for each process p in g, send (p,m).

    receive(m): B-deliver(m) at p.
- Guarantees: message is eventually delivered to the group if:
  - Processes are non-faulty.
  - The unicast "send" is reliable.
  - *Sender does not crash.*
- *Can we provide reliable delivery even after sender crashes?*
  - *What does this mean?*

Reliable Multicast (R-Multicast)

- **Integrity**: A *correct* (i.e., non-faulty) process *p* delivers a message *m* at most once.
  - *Assumption: no process sends **exactly** the same message twice*
- **Validity**: If a *correct* process multicasts (sends) message *m*, then it will eventually deliver *m* to itself.
  - *Liveness for the sender.*
- **Agreement**: If a *correct* process delivers message *m*, then all the other *correct* processes in group(*m*) will eventually deliver *m*.
  - *All or nothing.*
- Validity and agreement together ensure overall liveness: if some correct process multicasts a message *m*, then, all correct processes deliver *m* too.

How to achieve R-multicast? To be continued in next class….